Fondamenti d'Informatica

Corsi di Laurea in Ing. Civile e Ing. per l'Ambiente e il Territorio

Docente: Giorgio Fumera

https://www.unica.it/unica/page/it/giorgio_fumera_mat_fondamenti_dinformatica_1

Università degli Studi di Cagliari
A.A. 2025/2026

Argomenti del corso

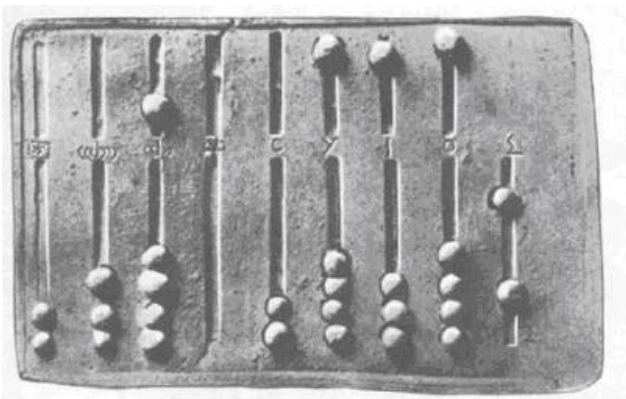
- Organizzazione e funzionamento dei calcolatori, sistema operativo
- Codifica binaria dell'informazione
- Algoritmi: formulazione, rappresentazione, proprietà
- Linguaggi e ambienti di programmazione
- Il linguaggio di programmazione Python

Organizzazione e funzionamento dei calcolatori, sistema operativo

Evoluzione dei calcolatori: cenni storici

I primi strumenti di calcolo

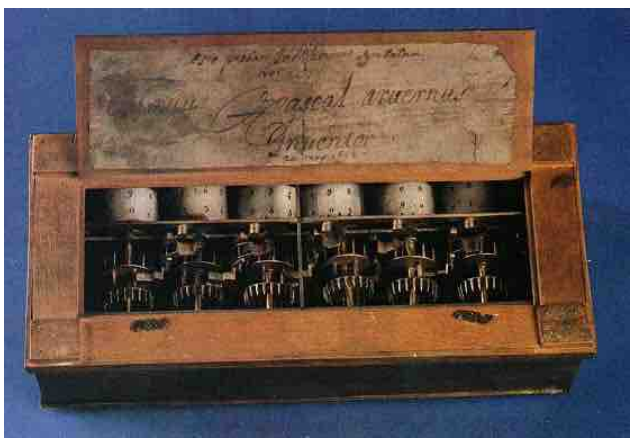
- tacche su legno o ossa
- sassi ("calcolo": dal latino *calculus*, sassolino)
- abaco (c. XXVIII secolo a.C.)



Evoluzione dei calcolatori: cenni storici

Le prime macchine calcolatrici **meccaniche**

"pascalina"
(B. Pascal, 1623–1662):
addizioni e sottrazioni

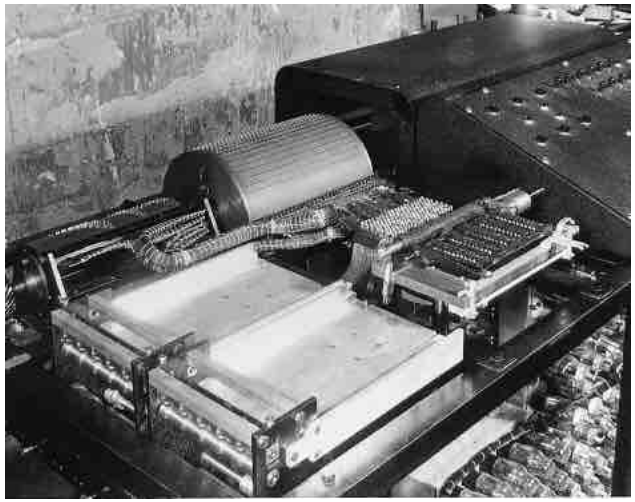


calcolatrici da ufficio
(inizi del '900)

Evoluzione dei calcolatori: cenni storici

Anni '30: primi calcolatori **elettromeccanici** in grado di svolgere **un solo** tipo di calcolo, per esempio:

- costruzione di tabelle di tiro per l'artiglieria
- risoluzione di sistemi di equazioni lineari

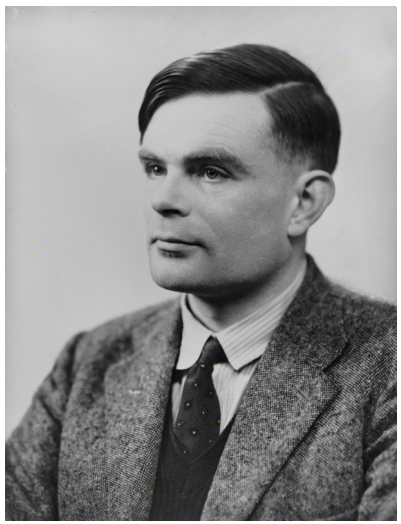


Procedimento (**algoritmo**) **codificato** nei meccanismi o circuiti:

Evoluzione dei calcolatori: cenni storici

Anni '30: contributi dalla matematica

- analisi dei procedimenti di calcolo (**algoritmi**) e della possibilità di una loro automatizzazione
- idea di macchina **programmabile**, esecutrice automatica di algoritmi (calcolatore **universale**)
- organizzazione **logica** di un calcolatore programmabile



Alan M. Turing (1912-1954)

John von Neumann (1903-1957)

Evoluzione dei calcolatori: cenni storici

Algoritmo:

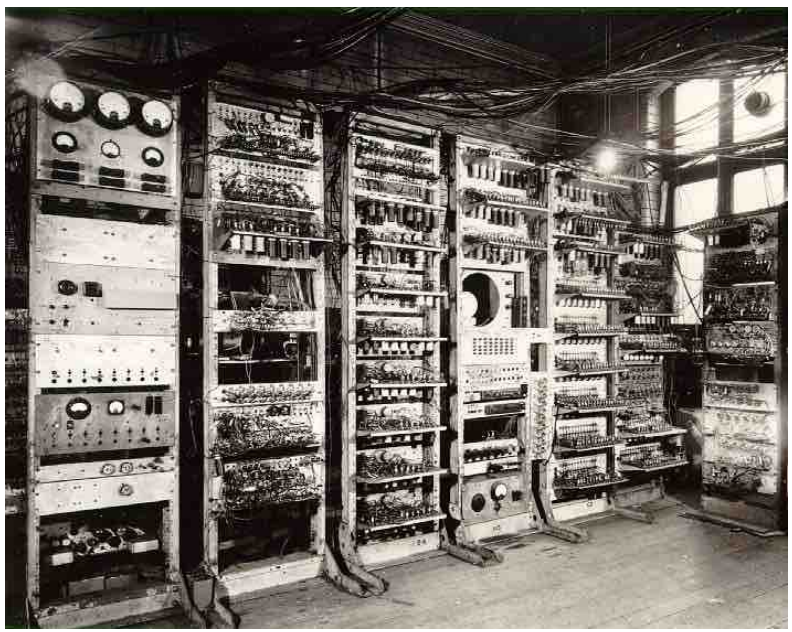
descrizione del procedimento per l'esecuzione di una data operazione, espressa in modo non ambiguo, in un linguaggio ***comprensibile*** *da un dato* ***esecutore****, in termini di una sequenza* ***finita*** *di azioni, ciascuna delle quali sia* ***eseguibile*** *dall'esecutore.*

Macchina esecutrice di algoritmi (calcolatore **programmabile**):

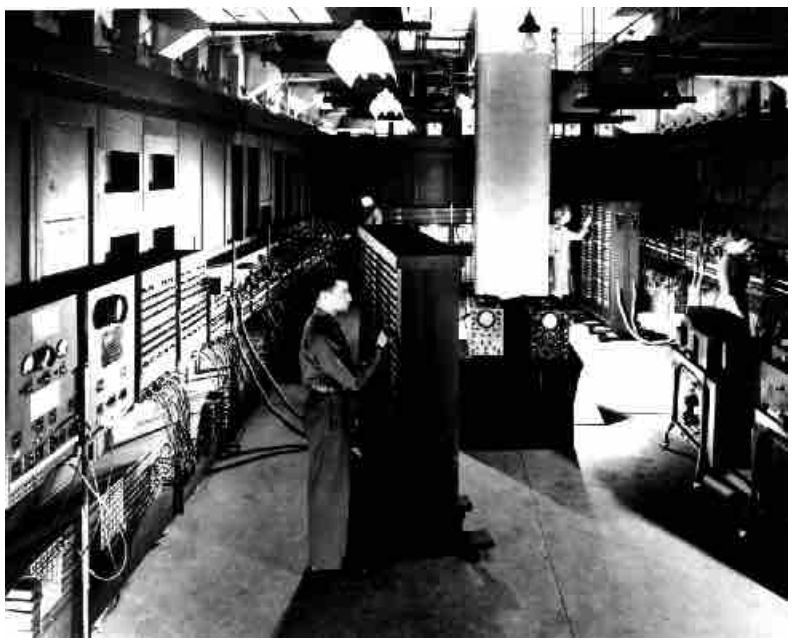
Evoluzione dei calcolatori: cenni storici

Anni '40: la tecnologia elettronica rende possibile la realizzazione di calcolatori **programmabili**.

Alcuni dei primi calcolatori:



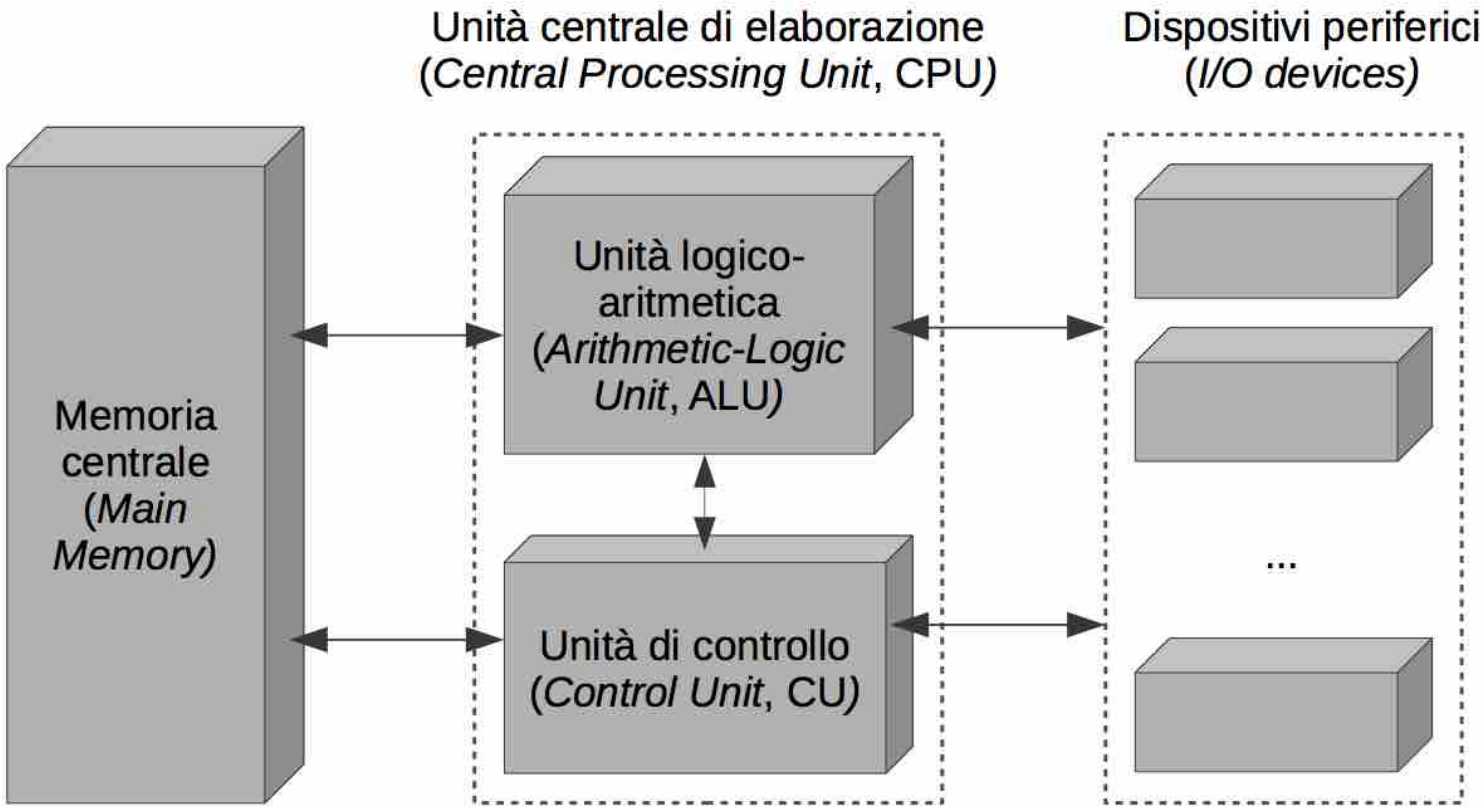
Mark I (1944)
(calcolatore **elettromeccanico**)



ENIAC (1946)
(il primo calcolatore **elettronico**)

Architettura di Von Neumann

Organizzazione **logica** di un calcolatore **programmabile**
(J. von Neumann, 1945)

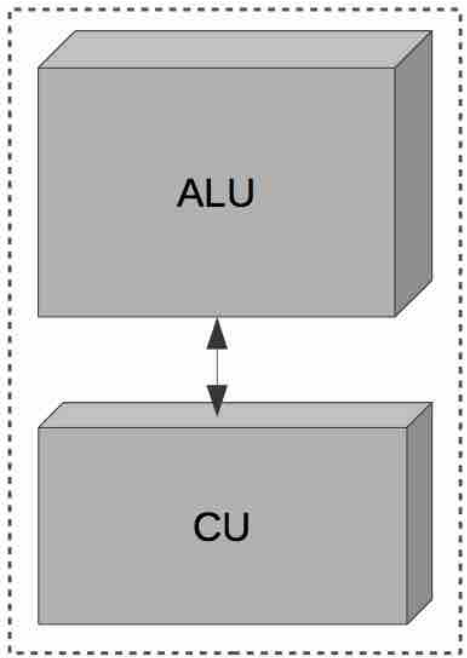


Memoria centrale

- Sequenza di **celle** o **parole** identificate univocamente da un **indirizzo** numerico
- Ciascuna cella memorizza un numero di **bit** predefinito
- Funzione: memorizzare il **programma in esecuzione** e i **dati** che esso dovrà elaborare
- Unità di misura della capacità di memoria: **byte** (8 bit) e suoi multipli: kilobyte (KB), megabyte (MB), ecc.

Unità centrale di elaborazione

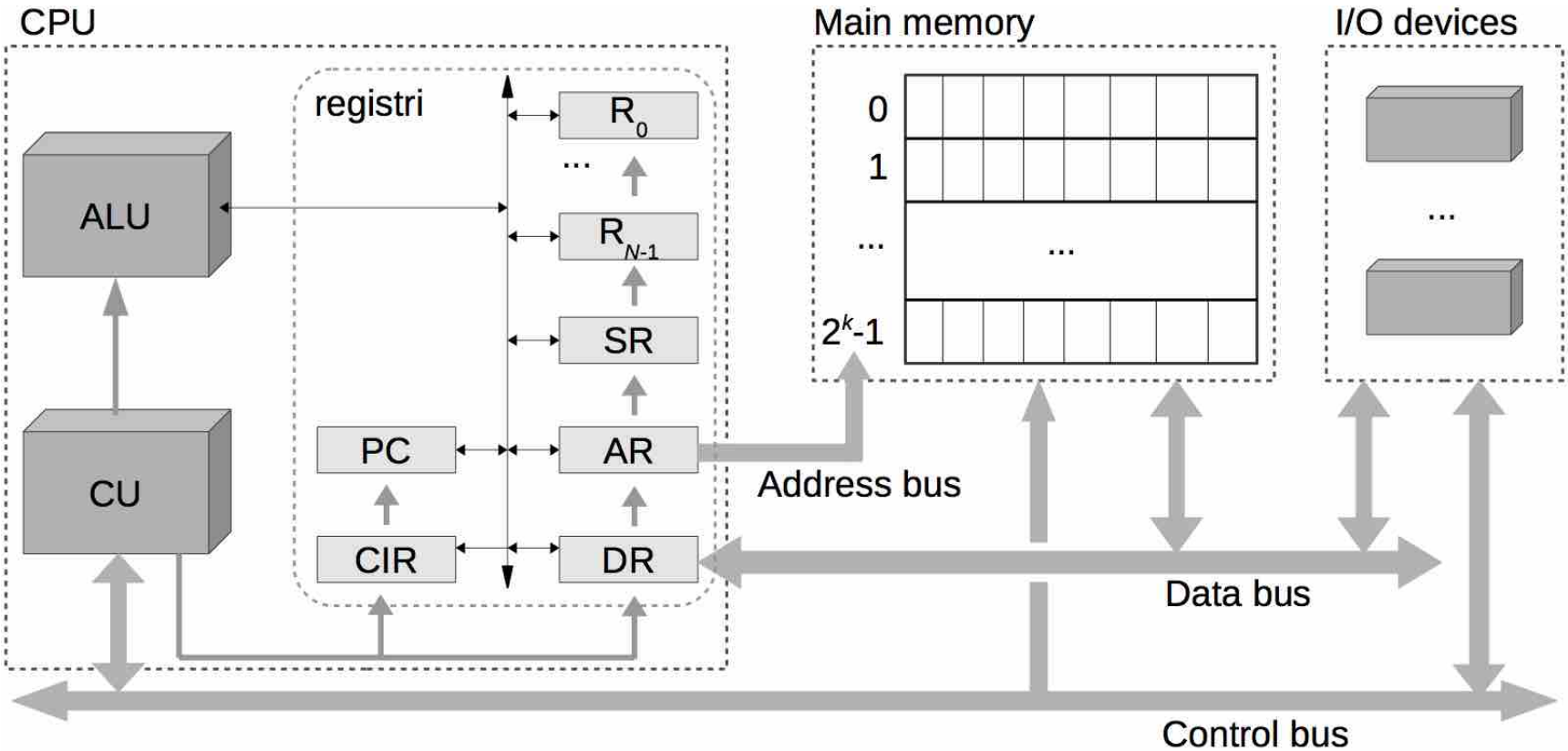
- Unità logico-aritmetica (ALU)
- Unità di controllo (CU)
- Registri:
  - *program counter* (PC)
  - *current instruction register* (CIR)
  - *address register* (AR)
  - *data register* (DR)
  - *status register* (SR)
  - registri di lavoro
- Orologio (*clock*) di sistema



Bus di sistema

- *Bus* **dati**
- *Bus* **indirizzi**
- *Bus* **controlli**

Architettura di un calcolatore: dettagli



Programmi

Programma: sequenza d'istruzioni codificate in forma binaria

Nell'architettura di Von Neumann

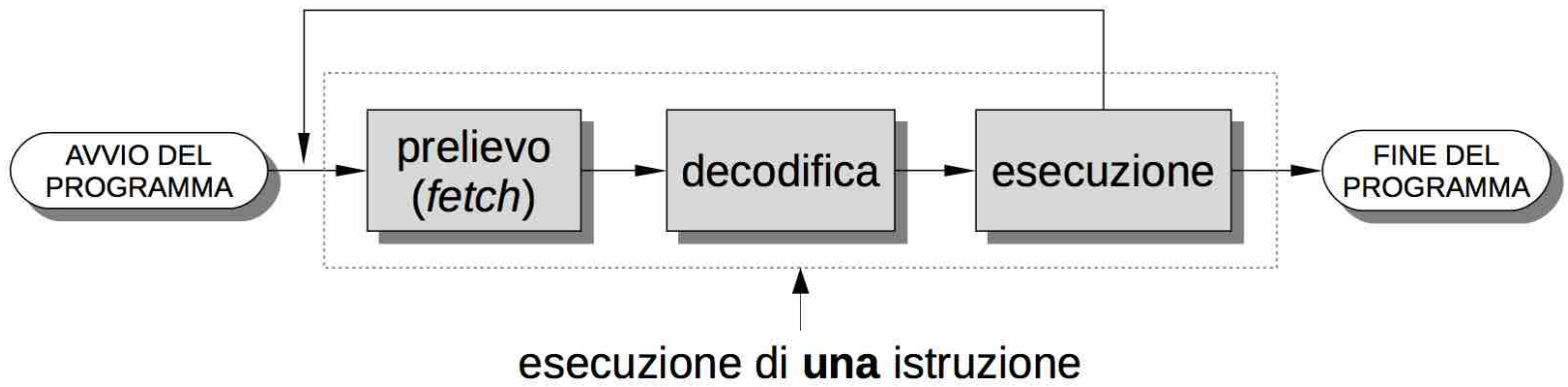
- un calcolatore esegue **un solo** programma alla volta
- ogni istruzione è memorizzata in **una** cella di memoria
- la sequenza d'istruzioni è memorizzata in celle **adiacenti**
- le istruzioni vengono eseguite **sequenzialmente** dalla CPU, nello stesso ordine nel quale sono memorizzate

Insieme d'istruzioni eseguibili dalla CPU

Poche decine di istruzioni, dette **linguaggio macchina**, suddivise in quattro categorie

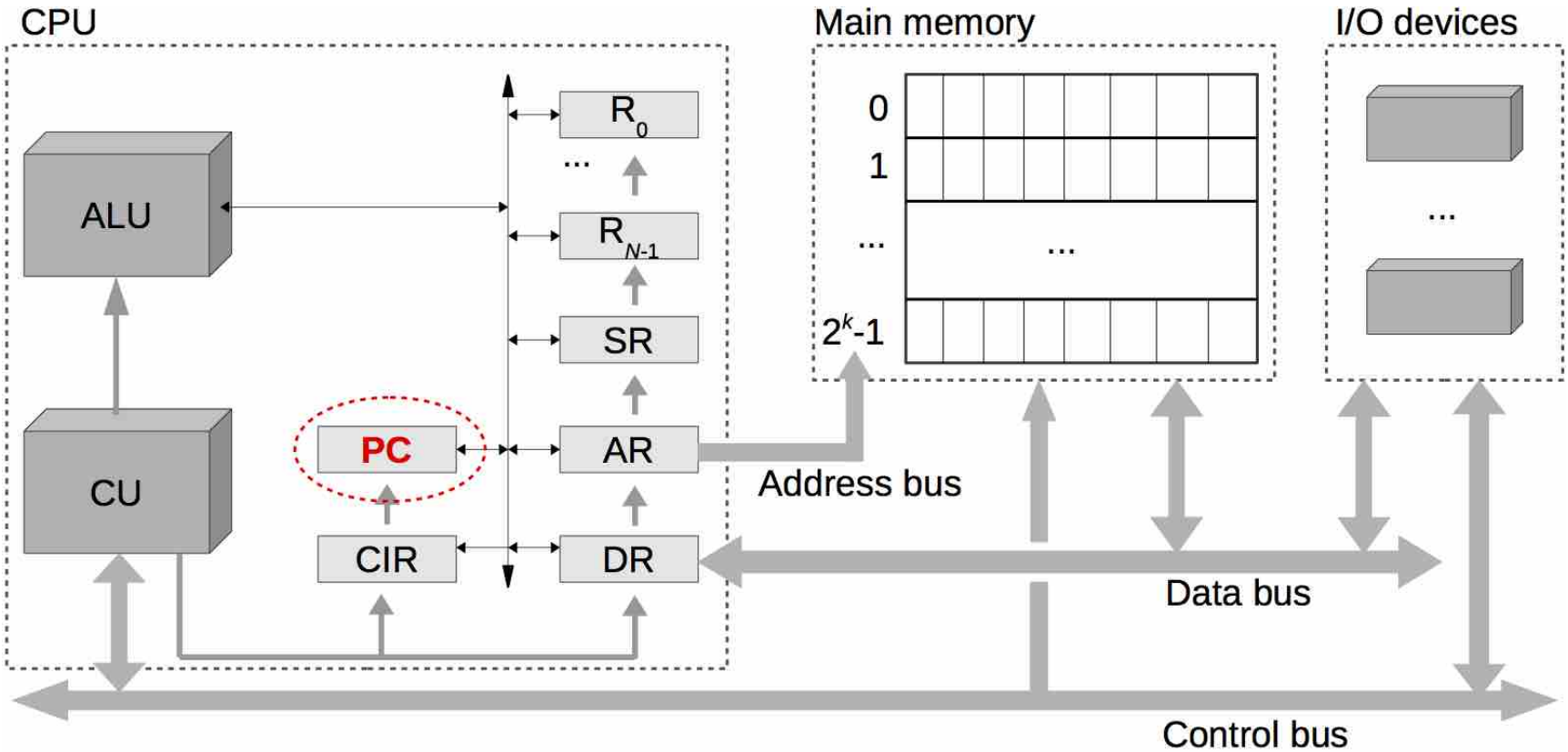
- elaborazione
- memorizzazione
- trasferimento
- controllo

Esecuzione delle istruzioni



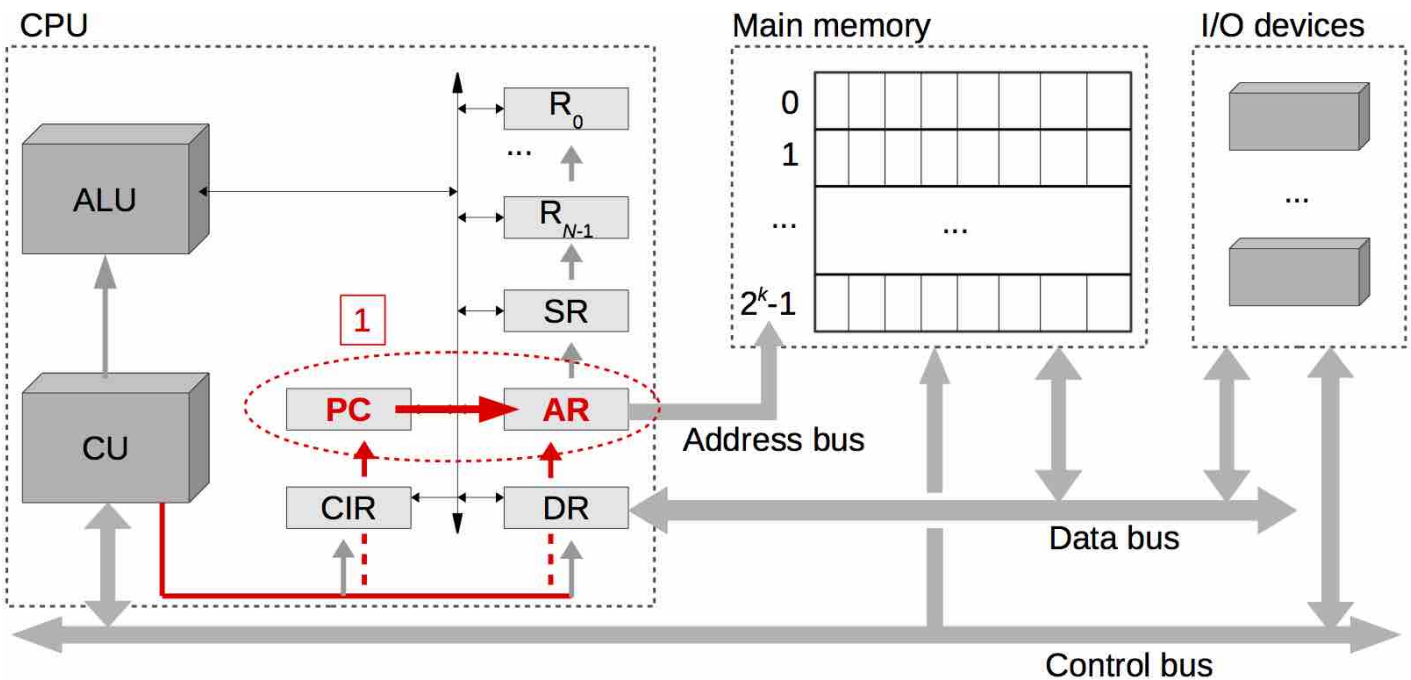
Avvio di un programma

Si memorizza nel PC l'indirizzo di memoria della **prima** istruzione



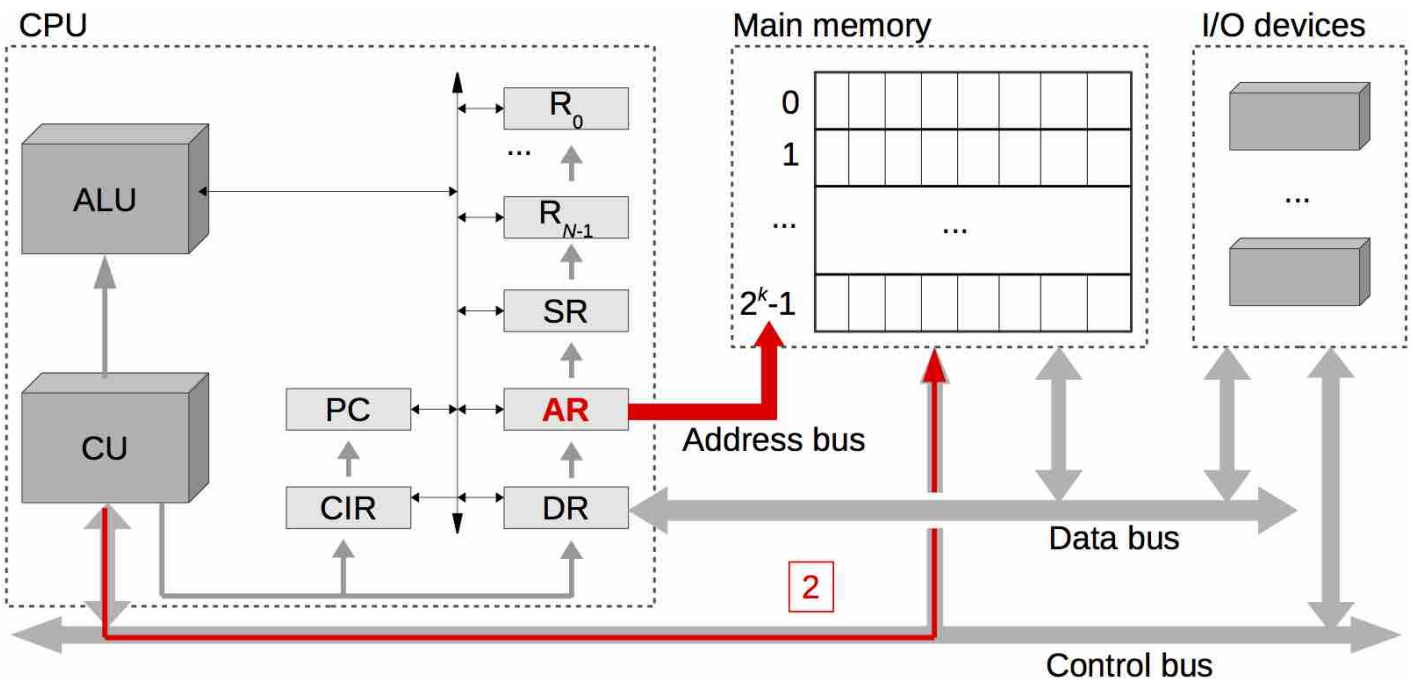
Esecuzione delle istruzioni: fase di prelievo

1. AR ← PC il contenuto del PC viene copiato nell'AR



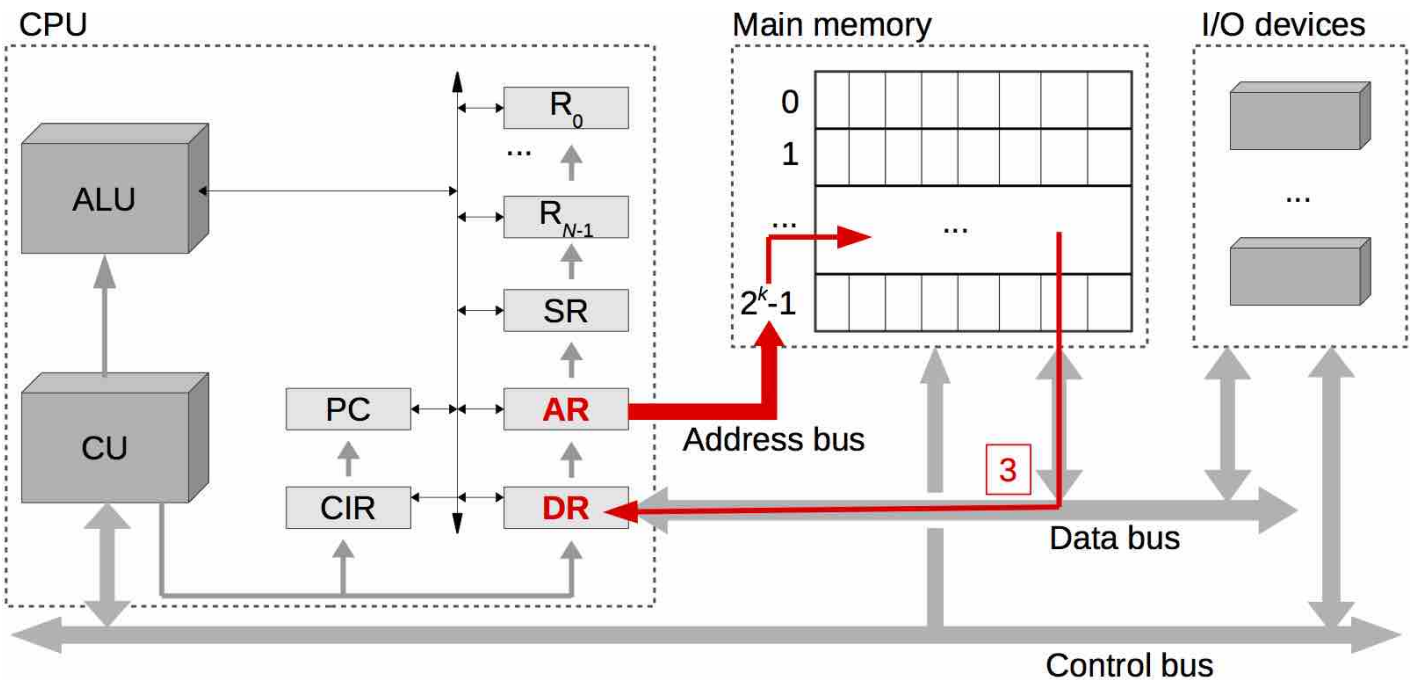
Esecuzione delle istruzioni: fase di prelievo

1. $\texttt{AR} \leftarrow \texttt{PC}$ il contenuto del PC viene copiato nell'AR
2. CU $\overset{\text{READ}}{\rightarrow}$ M la CU chiede alla memoria un'operazione di **lettura**



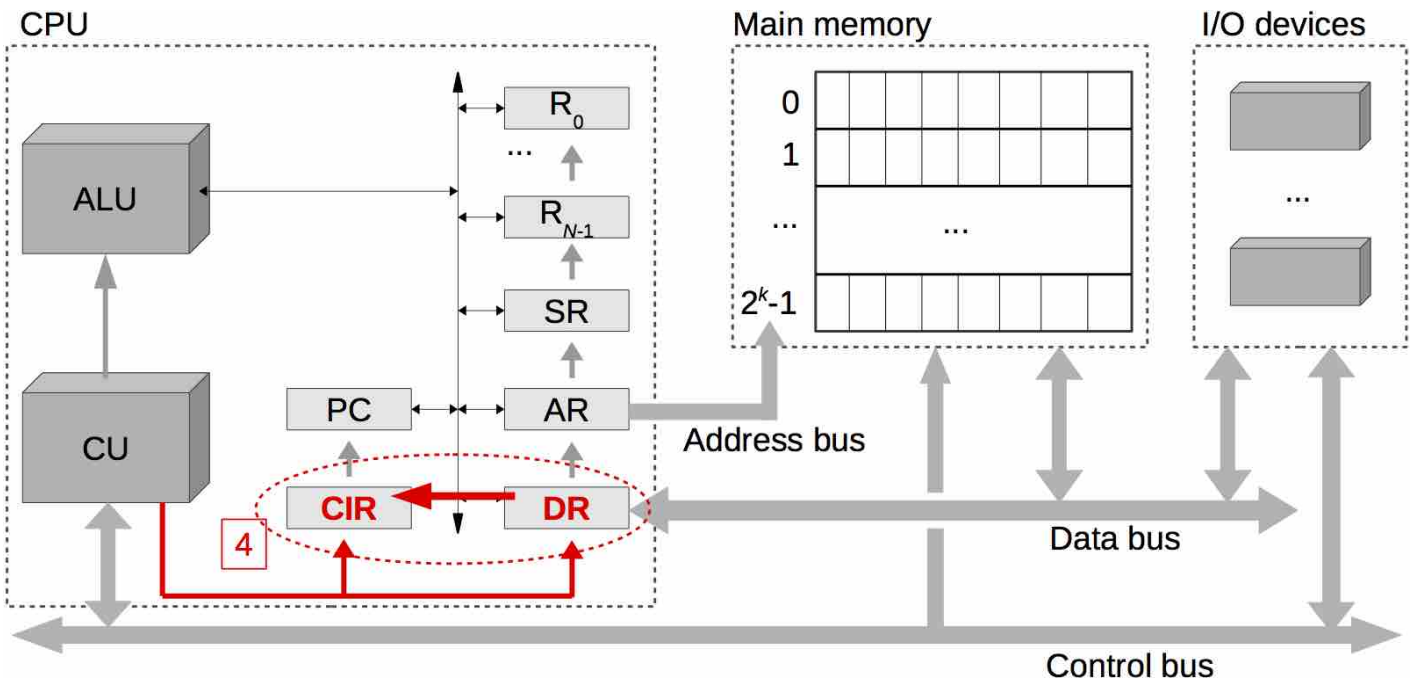
Esecuzione delle istruzioni: fase di prelievo

1. $\texttt{AR} \leftarrow \texttt{PC}$ il contenuto del PC viene copiato nell'AR
2. CU $\overset{\text{READ}}{\rightarrow}$ M la CU chiede alla memoria un'operazione di **lettura**
3. $\texttt{DR} \leftarrow \texttt{M[AR]}$ il contenuto della cella il cui indirizzo si trova nell'AR viene copiato del DR



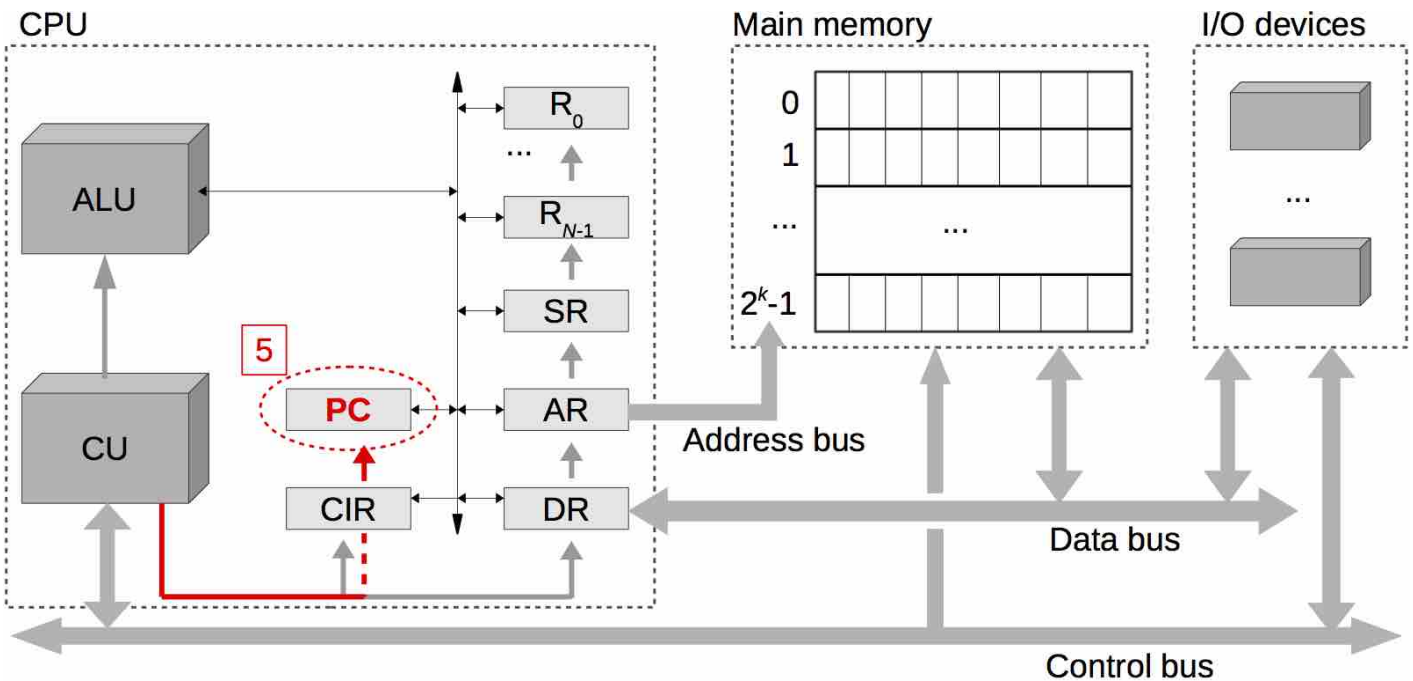
Esecuzione delle istruzioni: fase di prelievo

1. $\texttt{AR} \leftarrow \texttt{PC}$ — il contenuto del PC viene copiato nell'AR
2. $\text{CU} \overset{\text{READ}}{\rightarrow} \text{M}$ — la CU chiede alla memoria un'operazione di **lettura**
3. $\texttt{DR} \leftarrow \text{M}[\texttt{AR}]$ — il contenuto della cella il cui indirizzo si trova nell'AR viene copiato del DR
4. $\texttt{CIR} \leftarrow \texttt{DR}$ — il contenuto del DR viene copiato nel CIR



Esecuzione delle istruzioni: fase di prelievo

1. $\texttt{AR} \leftarrow \texttt{PC}$ — il contenuto del PC viene copiato nell'AR
2. $\text{CU} \overset{\text{READ}}{\rightarrow} \text{M}$ — la CU chiede alla memoria un'operazione di **lettura**
3. $\texttt{DR} \leftarrow \text{M}[\texttt{AR}]$ — il contenuto della cella il cui indirizzo si trova nell'AR viene copiato del DR
4. $\texttt{CIR} \leftarrow \texttt{DR}$ — il contenuto del DR viene copiato nel CIR
5. $\texttt{PC} \leftarrow \texttt{PC} + 1$ — si memorizza nel PC l'indirizzo della **prossima** istruzione



Evoluzione dei calcolatori

Evoluzione della tecnologia elettronica

- valvole termoioniche (1945–1955)
- *transistor* (1955–1965)
- circuiti integrati (1965–1980)
- circuiti ad alta densità di integrazione (1980–oggi)

Principali conseguenze

- diminuzione delle dimensioni, del consumo energetico e del costo di fabbricazione dei calcolatori
- incremento del numero di *transistor* per unità di superficie, della velocità di elaborazione e della capacità dei dispositivi di memoria
- estensione dei campi di applicazione dei calcolatori

Estensioni dell'architettura di von Neumann

Due caratteristiche dell'architettura di von Neumann limitano la velocità di esecuzione dei programmi

- esecuzione **sequenziale** delle istruzioni di un programma, anche se alcune di esse potrebbero essere eseguite **in parallelo**
- necessità di **prelevare** le istruzioni (e i dati da elaborare) dalla memoria centrale: tale operazione richiede tempi superiori rispetto all'esecuzione di un'istruzione logico-aritmetica

La velocità di esecuzione dei programmi dipende anche da altri fattori, tra i quali

- frequenza dell'orologio di sistema
- complessità delle istruzioni del linguaggio macchina
- ampiezza dei registri, del *bus* dati e del *bus* indirizzi
- capacità di memoria
- tempo d'accesso alla memoria

Estensioni dell'architettura di von Neumann

L'architettura di von Neumann è stata pertanto modificata in vari modi nel corso degli anni per aumentarne l'efficienza.

Modifiche principali

- introduzione di diversi dispositivi di memoria
- realizzazione di CPU in grado di eseguire diverse istruzioni simultaneamente
- uso di più CPU per eseguire diverse istruzioni simultaneamente

Il sistema di memoria

Caratteristiche principali di un dispositivo di memoria

- capacità
- costo per bit
- tempo di accesso
- dimensioni
- consumo di energia
- memorizzazione permanente o meno dei dati
- possibilità di trasporto tra diversi calcolatori
- ...

Nessun dispositivo fisico può raggiungere un compromesso accettabile tra le diverse caratteristiche.

Gerarchia di memorie

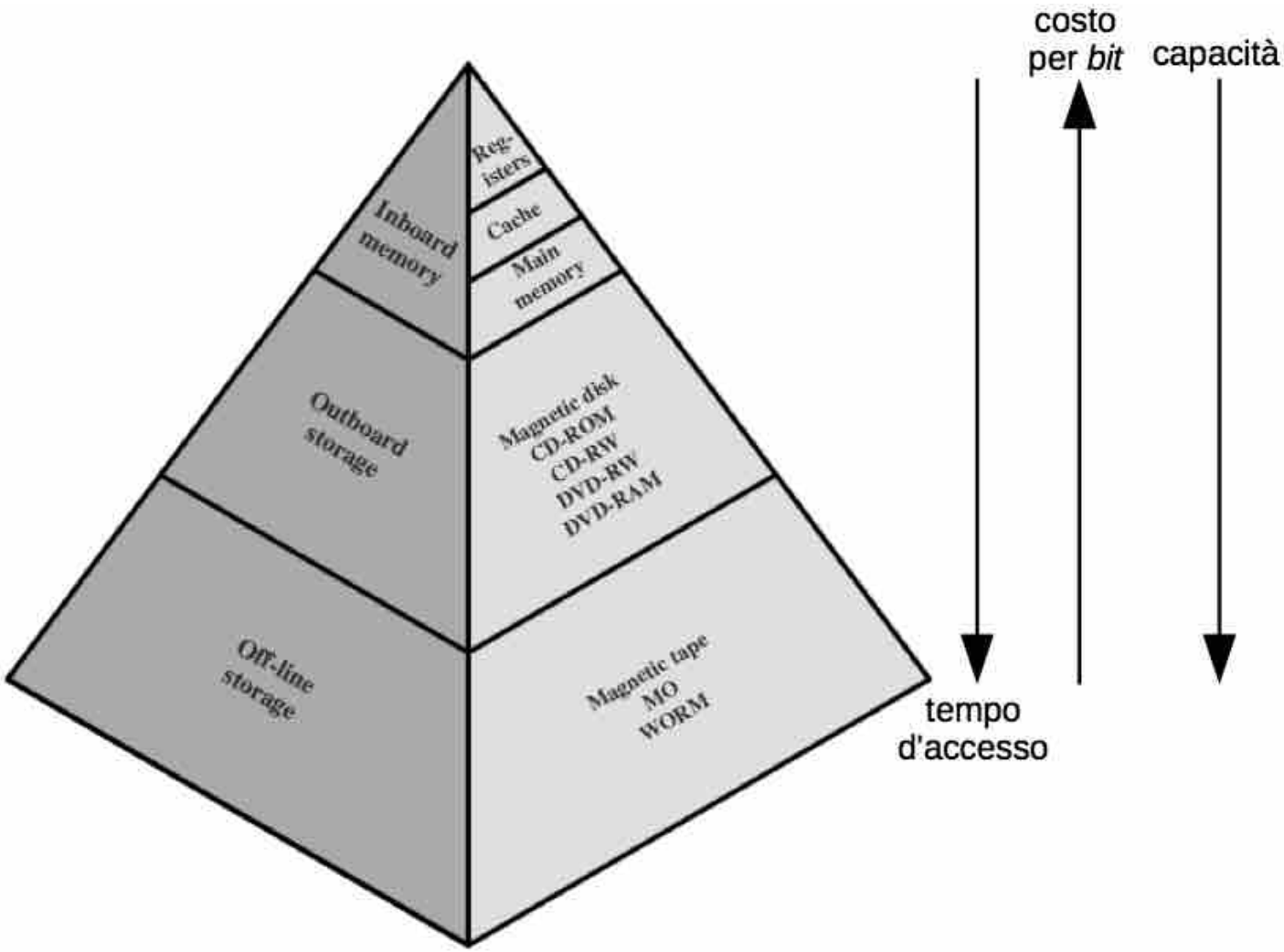
Soluzione: introduzione di **diversi** dispositivi di memoria caratterizzati dal **compromesso** tra

- tempo d'accesso
- costo (per bit)
- capacità

Organizzazione **gerarchica** dei diversi dispositivi per tempo d'accesso e capacità **crescenti**, e costo per bit **decrescente**

- registri della CPU
- memoria *cache*
- **memoria centrale**
- memoria secondaria (*outboard storage*)
- memoria terziaria (*outboard storage* e *off-line storage*)

Gerarchia di memorie



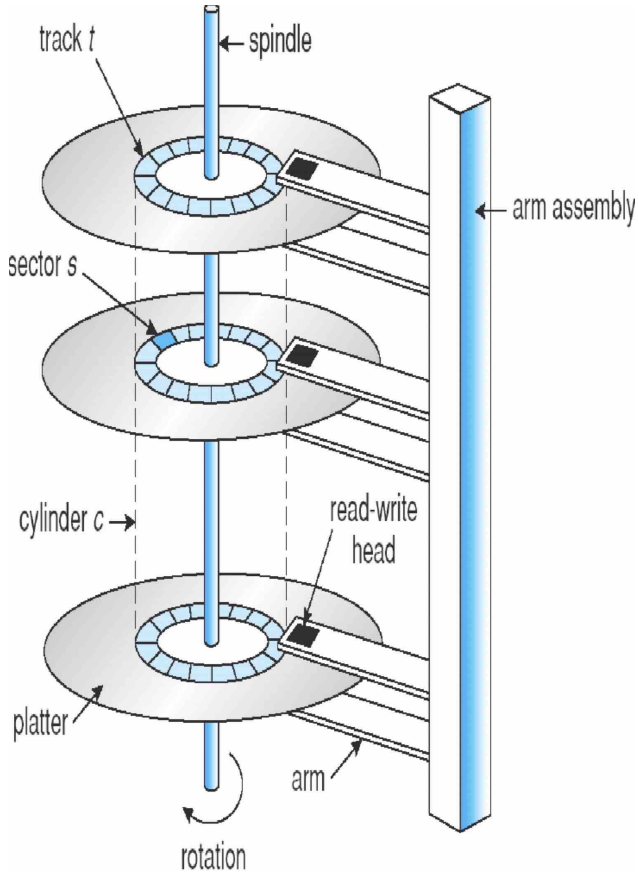
Memoria centrale

- Obiettivo principale: riduzione del **tempo d'accesso**, rispetto al tempo necessario alla CPU per eseguire un'istruzione logico-aritmetica
- Conseguenze
  - tecnologia: circuiti integrati (analogamente alla CPU, per ridurre il tempo d'accesso)
  - elevato costo per bit (a causa della tecnologia scelta)
  - capacità limitata (per ridurre il costo totale)
  - volatilità (per ridurre il tempo d'accesso)

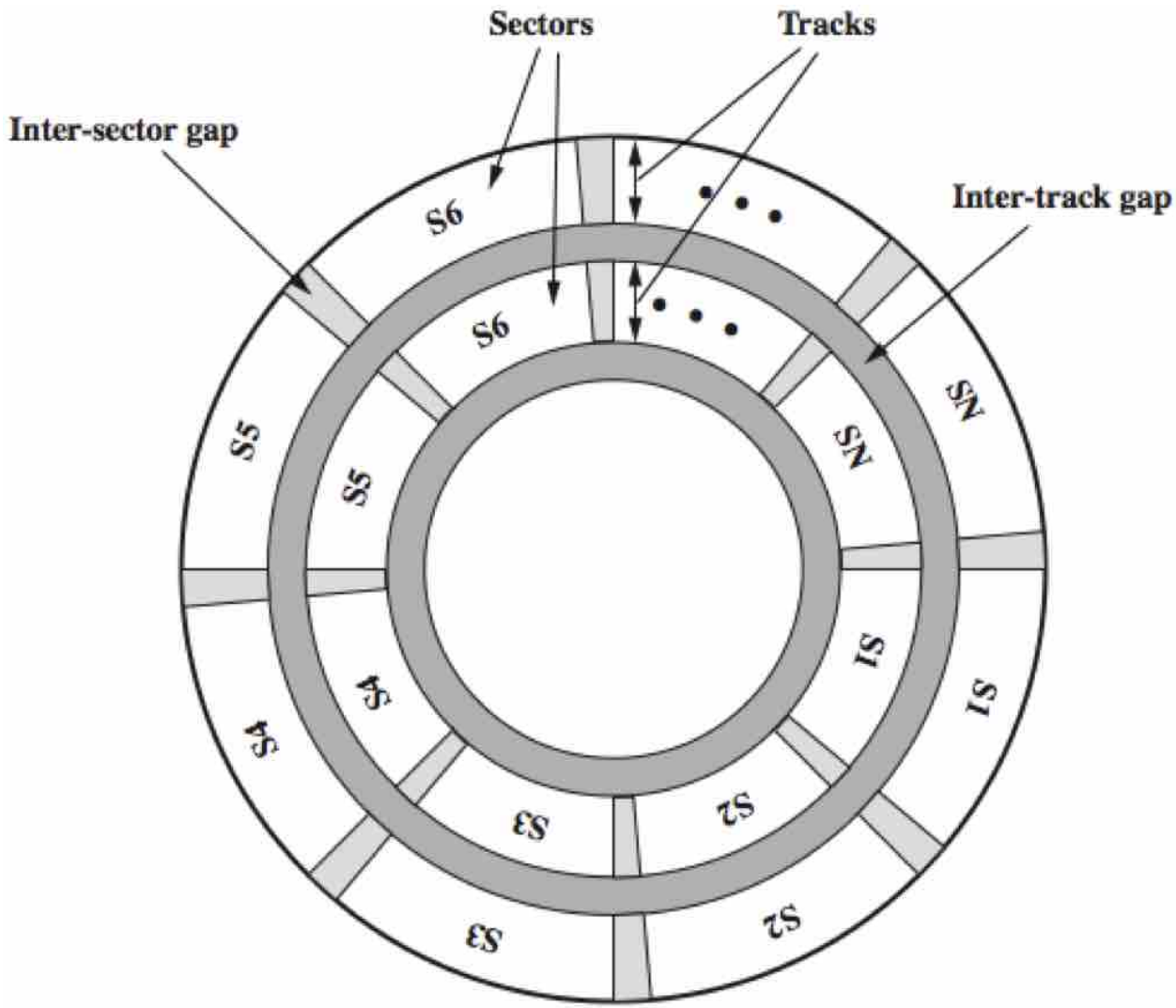
Memoria secondaria

- Obiettivi principali
  - non volatilità
  - collegamento permanente con la CPU
  - capacità elevata
  - basso costo per bit
- Conseguenze
  - tecnologia: dischi magnetici – *hard disk* –, e più recentemente circuiti integrati – *solid state drive* – (non volatilità, riduzione del costo per bit)
  - tempo d'accesso elevato (a causa della tecnologia scelta)

Memoria secondaria: dischi magnetici



Memoria secondaria: dischi magnetici



Memoria terziaria

- Obiettivi principali
  - non volatilità
  - facilità di collegamento e rimozione dal calcolatore, per es. per il trasferimento di dati tra diversi calcolatori, e per realizzare copie di sicurezza dei dati (**backup**)
  - capacità elevata
  - basso costo per bit
- Conseguenze
  - tecnologia: nastri e dischi magnetici, dischi ottici – CD-ROM, CD-RW, DVD –, circuiti integrati – per es., USB *flash drive* – (non volatilità, riduzione del costo per bit)
  - tempo d'accesso elevato (a causa della tecnologia scelta)

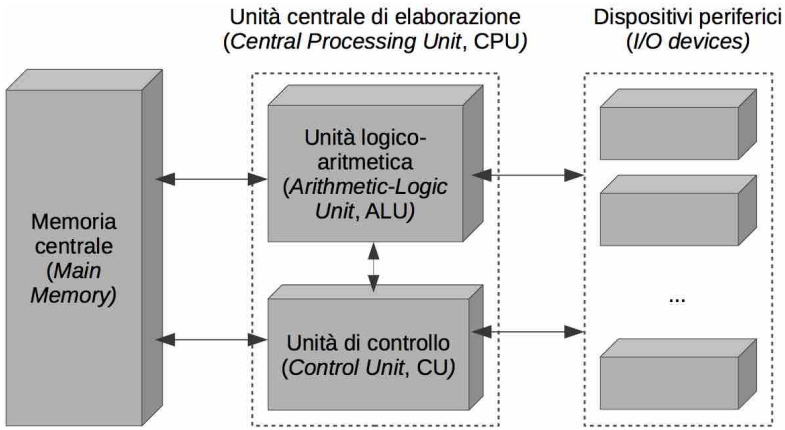
Memoria *cache*

- Obiettivo: basso tempo d'accesso, per sopperire alla crescente divergenza tra il tempo d'esecuzione delle istruzioni logico-aritmetiche da parte della CPU e il tempo d'accesso alla memoria principale
- Conseguenze
  - tecnologia: circuiti integrati (analogamente alla CPU, per ridurre il tempo d'accesso)
  - costo per bit molto elevato (a causa della tecnologia scelta)
  - capacità molto limitata (per ridurre il costo totale)
  - volatilità (per ridurre il tempo d'accesso)

Sistemi operativi: cenni storici

Calcolatore: insieme di **risorse fisiche** (***hardware***) necessarie per l'esecuzione di programmi

- memoria centrale
- CPU
- dispositivi periferici



Nei primi calcolatori elettronici (anni '40-'50) gli utenti avevano il totale controllo di tutti i dispositivi *hardware*, che dovevano quindi conoscere in ogni dettaglio.

Sistemi operativi: cenni storici

Primi anni '50: introduzione di programmi *di sistema* dedicati alla gestione di operazioni ripetitive, sollevando gli utenti da tale onere

- automatizzare le operazioni necessarie per l'esecuzione dei programmi (caricamento nella memoria centrale, avvio)
- gestione dei dispositivi periferici

Anni '50: sviluppo di nuovi programmi di sistema per rendere più efficiente e più semplice l'uso dei calcolatori, attraverso le seguenti funzionalità

- esecuzione simultanea di più programmi (*multiprogrammazione*)
- interazione con gli utenti (*time sharing*)
- interfacce utente (testuali, grafiche)

Tali programmi hanno dato origine ai *sistemi operativi*.

Sistema operativo

Insieme di programmi (***software* di sistema**) che gestiscono le risorse *hardware* di un calcolatore

- garantendone l'uso **corretto**, **efficiente** e **sicuro**
- fornendo un'**interfaccia** semplificata verso le risorse fisiche e logiche del calcolatore agli utenti e ai loro programmi

Funzioni del sistema operativo

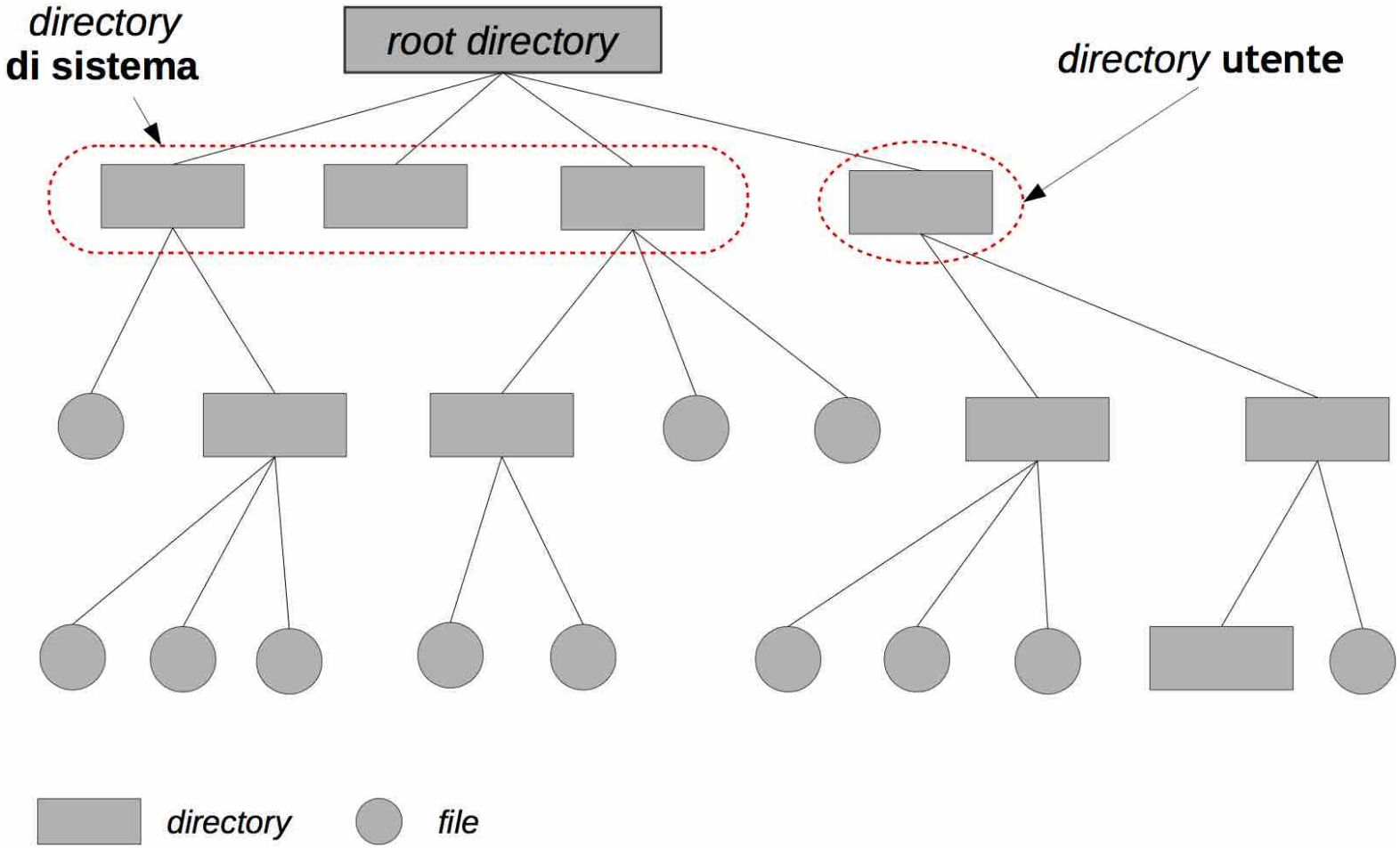
- Gestione dei **processi** (programmi in esecuzione): assegnazione della CPU a diversi programmi in esecuzione simultaneamente
- Gestione della **memoria centrale**:
  - assegnazione (**allocazione**) di un'area di memoria alle istruzioni e ai dati di ogni programma in esecuzione
  - **protezione** delle celle associate a ogni programma dall'accesso da parte di altri programmi
  - **memoria virtuale**: esecuzione simultanea di programmi che eccederebbero la capacità della memoria principale, mantenendo nella memoria secondaria una parte delle loro istruzioni e dei loro dati

(cont.)

Funzioni del sistema operativo

- Gestione logica della memoria secondaria e terziaria: ***file system***
  - ***file*** e ***directory*** (cartelle) come "contenitori" logici dei dati
  - organizzazione **gerarchica** (ad albero) delle cartelle
  - associazione di proprietà/attributi (data di creazione, utente proprietario, ecc.) e operazioni eseguibili (lettura, modifica, esecuzione) a ogni *file* e cartella
- Gestione dei dispositivi periferici
- Interfaccia utente (***shell***)
  - testuale ("a riga di comando")
  - grafica *Graphical User Interface*, GUI), dagli anni '80

Esempio: organizzazione gerarchica del *file system*



Codifica binaria dell'informazione

Sommario

- Codifica analogica e numerica
- Richiami sui sistemi posizionali e sulle basi di numerazione
- Codifica binaria dei numeri
  - numeri naturali
  - numeri interi: segno e valore, complemento a due
  - numeri reali: virgola fissa, virgola mobile

Introduzione

Dato: rappresentazione di un messaggio per mezzo di un sistema convenzionale di segni.
Informazione: il contenuto (significato) di un messaggio scambiato tra due interlocutori.

La memorizzazione, trasmissione ed elaborazione dell'informazione richiedono

- un supporto fisico
- una codifica

Tecniche di codifica dell'informazione

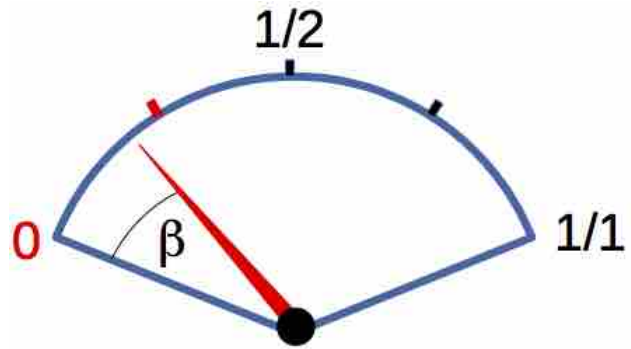
- **analogica**
- **numerica** (o "digitale")

Codifica analogica

Sfrutta grandezze fisiche variabili con **continuità**, in grado di assumere **infiniti** valori, e consente di riprodurre in modo esatto (in teoria) l'andamento di grandezze da codificare aventi anch'esse natura continua.

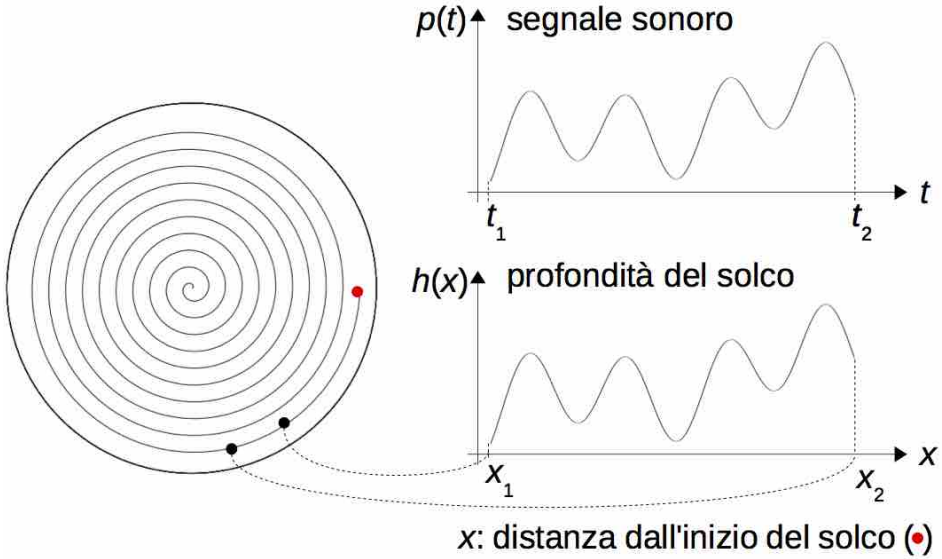
Esempio: livello del carburante in un'automobile

Un tipico indicatore analogico:



- valore da codificare: volume v (per es., in litri) del carburante presente nel serbatoio
- mezzo fisico: ago che ruota intorno a un asse fisso
- grandezza fisica: angolo β tra l'ago e il fondo scala
- codifica analogica: $\beta \propto v$ (ampiezza dell'angolo **proporzionale** al volume del carburante)

Esempio: codifica dei suoni nei dischi in vinile



- grandezza da codificare: segnale sonoro $p(t)$ (pressione)
- mezzo fisico: disco in vinile con un solco elicoidale sulla superficie
- grandezza fisica: profondità del solco, $h(x)$
- codifica: $t \to x$ (ogni punto del solco corrisponde a un istante di tempo), $h(x) \propto p(t)$ (la profondità nel punto x è **proporzionale** all'ampiezza del suono nell'istante t)

Codifica numerica

I valori della grandezza da codificare vengono associati a sequenze **discrete** e **finite** di **simboli** appartenenti a un **alfabeto** $\mathcal{A}$ composto da $k > 1$ elementi: $\mathcal{A} = \{s_0, s_1, \ldots, s_{k-1}\}$.

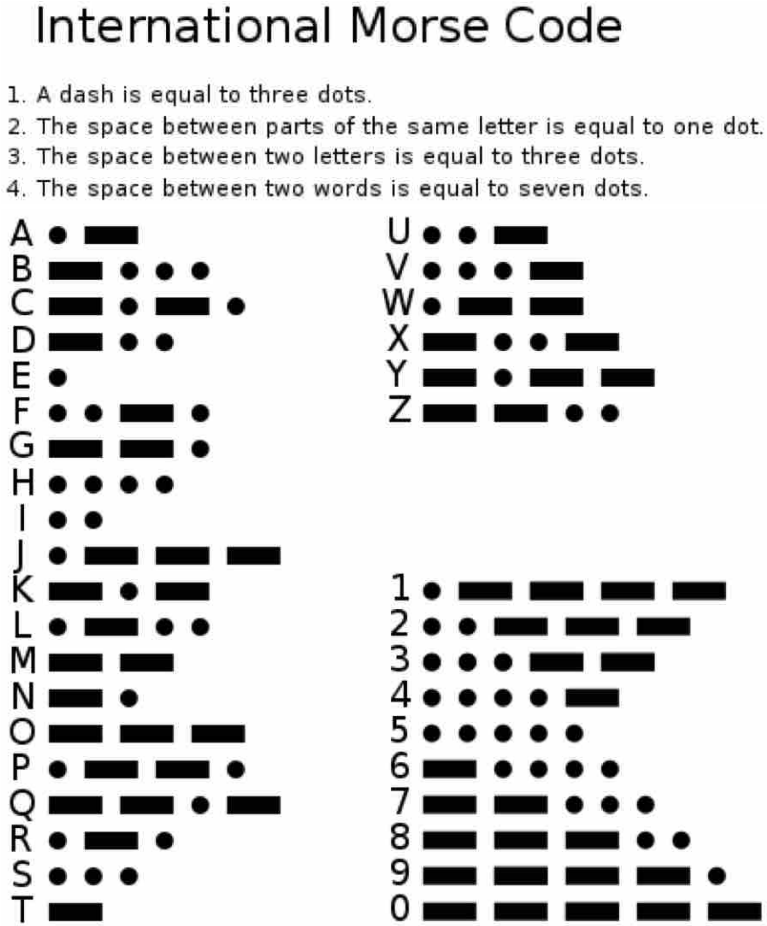
Ogni simbolo di $\mathcal{A}$ può essere visto come una **cifra**, e ogni sequenza di simboli come un **numero** espresso in base k.

In inglese, cifra $=$ *digit*: da qui il termine "codifica digitale".

Ciascun simbolo (cifra), o una sequenza di simboli, viene associato a una data configurazione del mezzo fisico usato per la codifica (per es., un valore di tensione in un dispositivo elettronico).

Esempio: il codice Morse

- Mezzo fisico: impulso elettrico, luminoso, ecc.
- Grandezza fisica: durata dell'impulso
- Equivale a una codifica numerica con $k = 4$
  - punto
  - linea
  - spazio tra due caratteri
  - spazio tra due parole



Nota: l'informazione da codificare ha essa stessa natura discreta e finita.

Codifica numerica

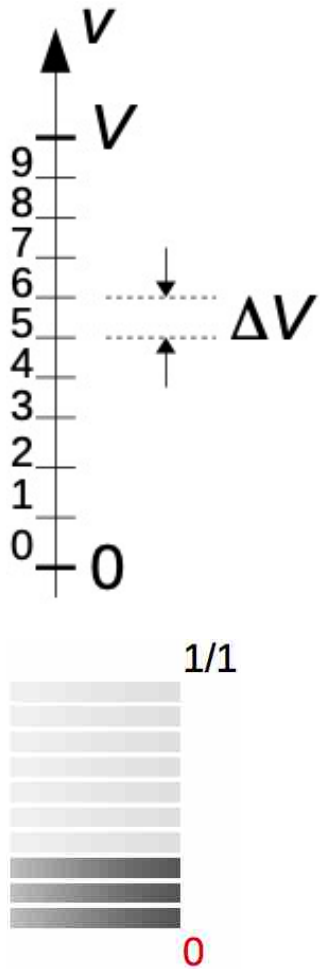
Se la grandezza X da codificare è di natura **continua**, è necessaria un'operazione preliminare di **discretizzazione**

- **quantizzazione**: l'intervallo $[X_{\min}, X_{\max}]$ dei valori che X può assumere viene suddiviso in un numero **finito** M di sottointervalli; ciascun valore all'interno di uno stesso sottointervallo viene codificato con un'**unica** sequenza di simboli, corrispondente a un numero tra 0 e $M-1$
- **campionamento**: se X varia in modo continuo nel tempo (per es., un suono) o nello spazio (per es., un'immagine), vengono misurati e codificati solo i valori (**campioni**) assunti da X in un sottoinsieme discreto e finito di istanti di tempo o punti dello spazio

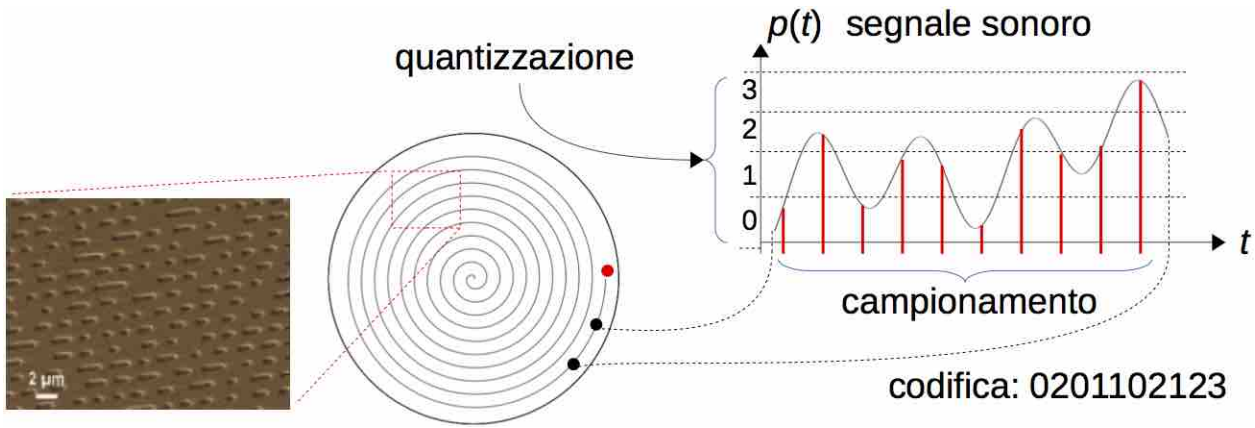
Esempio: livello del carburante in un'automobile

Tipico indicatore numerico: *display* a segmenti

- valore da codificare: volume v del carburante contenuto in un serbatoio di capacità V: $v \in [0, V]$
- **quantizzazione** in M intervalli (nell'esempio in figura, $M = 10$):
  - ampiezza di ogni intervallo: $\Delta L = \frac{V}{M}$
  - **qualsiasi** valore $v \in [m\Delta V,\ (m+1)\Delta V)$, $m = 0, \ldots, M-1$, viene codificato con il numero m
- mezzo fisico: *display* a M segmenti
- codifica: il valore m viene rappresentato dall'accensione dei primi $m+1$ segmenti (in figura un esempio per $m = 2$)



Esempio: codifica dei suoni nei *compact disc* (CD-ROM)



- grandezza da codificare: segnale sonoro $p(t)$ (pressione)
- **quantizzazione**: $M = 2^{16} = 65.536$ intervalli (in figura, $M = 4$)
- **campionamento**: 44.100 campioni/secondo
- mezzo fisico: disco in materiale plastico
- grandezza fisica e codifica: presenza/assenza di fori (*pit*/*land*) di dimensione predefinita lungo una linea elicoidale sulla superficie

Codifica numerica

Quantizzazione e campionamento causano una **perdita d'informazione**, assente (in principio) nella codifica analogica.

Ci sono tuttavia due vantaggi rispetto alla codifica analogica

- maggiore **robustezza al rumore**: è sufficiente che le configurazioni fisiche associate ai simboli (per es., *pit* e *land*) restino **distinguibili**
- possibilità di memorizzare, trasmettere ed elaborare **qualsiasi tipo** d'informazione (testi, suoni, immagini, ecc.) su **uno stesso** mezzo o dispositivo fisico

Codifica binaria

Un alfabeto $\mathcal{A}$ di $k = 2$ simboli porta a una codifica **binaria**.

Vantaggi rispetto a $k > 2$

- maggior robustezza al rumore
- maggior semplicità nella progettazione dei dispositivi di memorizzazione, trasmissione ed elaborazione

Alfabeto **convenzionale**: $\mathcal{A} = \{0,\ 1\}$.
I due simboli sono detti **bit**, da ***bi****nary digi****t*** (cifra binaria).

Qualsiasi sequenza finita di simboli binari può essere vista come la **rappresentazione** di un **numero naturale** in **base due**: questo rende conveniente l'uso della numerazione in base due per definire la codifica binaria di informazione di **qualsiasi** natura (testi, suoni, immagini, ecc.).

Cenni sulle basi di numerazione: sistemi posizionali

In una base $b \in \mathbb{N},\ b > 1$, un qualsiasi numero N è espresso come sequenza di cifre $c_i \in \{0, 1, \ldots, N-1\}$:

$$\underbrace{c_{n-1}c_{n-2}\ldots c_1 c_0}_{\text{parte intera}} \ , \ \underbrace{c_{-1}c_{-2}\ldots c_{-m}}_{\text{parte frazionaria}}$$

Il suo valore è definito come:

$$\begin{aligned} N = \quad & c_{n-1} \times b^{n-1} + c_{n-2} \times b^{n-2} + \ldots + c_1 \times b^1 + c_0 \times b^0 + \\ & c_{-1} \times b^{-1} + c_{-2} \times b^{-2} \ldots + c_{-m} \times b^{-m} \\ = \quad & \sum_{i=-m}^{n-1} c_i \times b^i \end{aligned}$$

Il **valore** di una cifra dipende dalla sua **posizione** nella sequenza

- la cifra avente peso maggiore (quella più a sinistra nella sequenza) è detta cifra **più significativa**
- quella avente peso minore è detta cifra **meno significativa**

Sistemi posizionali: un esempio

Per chiarezza, nell'espressione di un numero la base può essere indicata in lettere, come pedice.

Nelle espressioni seguenti, tutti i valori a destra del segno di uguaglianza sono scritti in base dieci

- $215,34_{\text{dieci}} = 2 \times 10^2 + 1 \times 10^1 + 5 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2}$
- $-215,34_{\text{sei}} = -(2 \times 6^2 + 1 \times 6^1 + 5 \times 6^0 + 3 \times 6^{-1} + 4 \times 6^{-2})$
- $110,01_{\text{due}} = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Sistemi non posizionali: un esempio

Un sistema di numerazione **non** posizionale: il sistema romano

- `I` $\rightarrow$ uno
- `V` $\rightarrow$ cinque
- `X` $\rightarrow$ dieci
- `L` $\rightarrow$ cinquanta
- `C` $\rightarrow$ cento
- `D` $\rightarrow$ cinquecento
- `M` $\rightarrow$ mille

Per esempio, in `XI` e in `XXIII` sia `I` che `X` hanno valore pari a uno, **indipendentemente** dalla loro posizione.

I valori di cifre adiacenti devono essere sommati o sottratti a seconda della posizione relativa.

Basi maggiori di dieci

Per basi $b > 10_{\text{dieci}}$, le cifre corrispondenti ai valori 10_{dieci}, $11_{\text{dieci}}, \ldots$ possono essere rappresentate mediante lettere, per es.

- $10_{\text{dieci}} \rightarrow$ A
- $11_{\text{dieci}} \rightarrow$ B
- ecc.

Per esempio, in base **sedici**:

$$2\text{A}0, \text{F}4_{\text{sedici}} = 2 \times 16^2 + \mathbf{10} \times 16^1 + 0 \times 16^0 + \mathbf{15} \times 16^{-1} + 4 \times 16^{-2}$$

Proprietà della rappresentazione posizionale

1. In qualsiasi base b, il valore $N = b$ si rappresenta come 10.
   Esempi:
   - $10_{\text{sette}} = 7_{\text{dieci}}$
   - $10_{\text{sedici}} = 16_{\text{dieci}}$
   - $10_{\text{due}} = 2_{\text{dieci}}$
2. Moltiplicare un numero N espresso in una qualsiasi base b per una qualsiasi potenza intera b^p produce un numero la cui espressione in base b si ottiene "spostando" di p posizioni a sinistra (se $p < 0$) o a destra (se $p > 0$) il separatore dei decimali.
   Dimostrazione: $\left(\sum_{i=-m}^{n-1} c_i \times b^i\right) \times b^p = \sum_{i=-m}^{n-1} c_i \times b^{i+p}$
   Esempi:
   - $-215,34_{\text{sei}} \times 10^{3_{\text{sei}}}_{\text{sei}} = -215340_{\text{sei}}$
   - $110,01_{\text{due}} \times 10^{-1_{\text{due}}}_{\text{due}} = 11,001_{\text{due}}$

Proprietà della rappresentazione posizionale

3. Si consideri un valore x espresso in una base b da una sequenza di $p \geq 0$ cifre intere e $q \geq 0$ cifre frazionarie tutte pari a $b-1$ (per es., in base dieci: $\underbrace{99\ldots9}_{p \text{ cifre}},\underbrace{99\ldots9}_{q \text{ cifre}}$).

   Tale valore è pari a $b^p - b^{-q}$.
   Dimostrazione:

$$\begin{aligned} x &= \sum_{i=-q}^{p-1}(b-1)\times b^i \\ &= \sum_{i=-q}^{p-1} b^{i+1} - \sum_{i=-q}^{p-1} b^i \\ &= \sum_{i=-q+1}^{p} b^i - \sum_{i=-q}^{p-1} b^i = b^p - b^{-q} \end{aligned}$$

   Esempio (in base dieci): $999,99 = 10^3 - 10^{-2} = 1000 - 0,01$.

   Casi particolari:

   - se x è intero ($q = 0$), allora $x = b^p - 1$
     (es.: $999_{\text{dieci}} = 1000_{\text{dieci}} - 1$)
   - se x ha parte intera nulla ($p = 0$), allora $x = 1 - b^{-q}$
     (es.: $0,999_{\text{dieci}} = 1_{\text{dieci}} - 0,001_{\text{dieci}}$)

Conversione in base dieci

Data la nostra familiarità con la base dieci, è facile convertire in essa numeri espressi in un'altra base: tutti i calcoli possono infatti essere eseguiti in base dieci.

Esempi:

$$\begin{aligned} -215,34_{\text{sei}} &= -(2\times 6^2 + 1\times 6^1 + 5\times 6^0 + 3\times 6^{-1} + 4\times 6^{-2}) \\ &= -(72 + 6 + 5 + \tfrac{3}{6} + \tfrac{4}{36}) \\ &= \ldots \end{aligned}$$

$$\begin{aligned} 110,01_{\text{due}} &= 1\times 2^2 + 1\times 2^1 + 0\times 2^0 + 0\times 2^{-1} + 1\times 2^{-2} \\ &= 4 + 2 + \tfrac{1}{4} \\ &= 6,25_{\text{dieci}} \end{aligned}$$

Conversione dalla base due alla base dieci

Per la base due esiste un procedimento più semplice: poiché in base due l'unica cifra diversa da zero ha valore uno, il valore di un numero si ottiene sommando le potenze di due corrispondenti alle cifre pari a uno.

Esempio: $110,01_{\text{due}}$

	parte intera			parte frazionaria	
cifre del numero	1	1	0	0	1
potenze di 2	$\mathbf{2^2}$	$\mathbf{2^1}$	2^0	2^{-1}	$\mathbf{2^{-2}}$
valore delle potenze	**4**	**2**	1	$1/2$	$\mathbf{1/4}$
valore in base dieci	$\mathbf{4} + \mathbf{2} + \mathbf{1/4} = 6,25$				

Conversione dalla base dieci a un'altra base

Usando lo stesso procedimento, il passaggio dalla base dieci a un'altra base b è meno intuitivo, poiché richiede di eseguire operazioni su valori espressi in base b.

Esempio: esprimere $-12,25_{\text{dieci}}$ in base tre.

Tenendo conto che $10_{\text{dieci}} = 101_{\text{tre}}$:

$$-12,25_{\text{dieci}} = -(\underbrace{1 \times 10^1 + 2 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}}_{\text{in base dieci}})$$

$$= -(\underbrace{1 \times \mathbf{101}^1 + 2 \times \mathbf{101}^0 + 2 \times \mathbf{101}^{-1} + \mathbf{12} \times \mathbf{101}^{-2}}_{\text{in base tre}})$$

$$= ?$$

Conversione dalla base dieci a un'altra base

Un procedimento alternativo consente di convertire un valore N dalla base dieci a un'altra base b, operando solo in base dieci.

Indicando con I e F la parte intera e quella frazionaria di N

- conversione della **parte intera**: **dividere** per b sia I che i quozienti ottenuti, fino a ottenere un quoziente pari a zero; le cifre di I in base b corrispondono alla sequenza dei **resti** così ottenuti (il primo corrisponde alla cifra **meno significativa**)
- conversione della **parte frazionaria**: **moltiplicare** per b sia F che la parte frazionaria dei prodotti ottenuti, fino a ottenere o un prodotto con parte frazionaria pari a zero, oppure il numero di cifre desiderato; le cifre di F in base b corrispondono alla sequenza delle parti intere di tali prodotti (la prima corrisponde alla cifra **più significativa**)

Esempio

Esprimere $-12,375_{\text{dieci}}$ in base due

- $I = 12_{\text{dieci}}$
- $F = 0,375_{\text{dieci}}$

Parte intera:

quozienti	resti
$I = 12$	**0**
6	**0**
3	**1**
1	**1**
0	

$I = \mathbf{1100}_{\text{due}}$

Parte frazionaria:

parti frazionarie	prodotti
$F =$ 0,375	**0**,75
0,75	**1**,5
0,5	**1**,**0**

$F = 0,\mathbf{011}_{\text{due}}$

Risultato: $-12,375_{\text{dieci}} = -1100,011_{\text{due}}$.

Conversione della parte frazionaria

Il procedimento di conversione per la parte frazionaria non avrebbe mai termine se N fosse irrazionale (per es., π), o se fosse periodico nella base d'interesse (per es., $\frac{1}{3}$ in base dieci). In questo caso si potrà terminare il procedimento dopo aver ottenuto un numero di cifre frazionarie pari alla precisione desiderata, o non appena il numero risulta essere periodico.

Esempio: esprimere $0{,}3_{\text{dieci}}$ in base due

parti frazionarie		prodotti
$F =$	0,3	**0**,6
	0,6	**1**,2
	0,2	**0**,4
	0,4	**0**,8
	0,8	**1**,6
	0,6	**1**,2

L'ultima parte frazionaria è identica a un'altra già ottenuta in precedenza (la seconda). Questo significa che il numero è periodico in base due:
$0{,}3_{\text{dieci}} = 0{,}0\overline{1001}_{\text{due}}$

Dimostrazione: parte intera

				quozienti	resti		
N	:	b	=	N_1	r_0	(1)	$N = N_1 \times b + r_0$
N_1	:	b	=	N_2	r_1	(2)	$N_1 = N_2 \times b + r_1$
N_2	:	b	=	N_3	r_2	(3)	$N_2 = N_3 \times b + r_2$
					...		
N_{n-2}	:	b	=	N_{n-1}	r_{n-2}	$(n-1)$	$N_{n-2} = N_{n-1} \times b + r_{n-2}$
N_{n-1}	:	b	=	$\mathbf{0}$	r_{n-1}	(n)	$N_{n-1} = r_{n-1}$

Sostituendo a ritroso l'espressione (n) nella $(n-1)$, l'espressione così ottenuta nella $(n-2)$, e così via fino alla (1), si ottiene:

$$N = r_{n-1} \times b^{n-1} + r_{n-2} \times b^{n-2} + \ldots + r_2 \times b^2 + r_1 \times b^1 + r_0 \times b^0$$

Poichè i valori r_i sono resti di divisioni per b, si ha che $r_i \in \{0, 1, \ldots, b-1\}$, $i = 0, \ldots, n-1$. Quindi, per definizione la sequenza $r_{n-1}\ r_{n-2}\ \ldots\ r_2\ r_1\ r_0$ è la rappresentazione di N in base b.

Dimostrazione: parte frazionaria

Sia $0, c_{-1}c_{-2}c_{-3}\ldots$ la rappresentazione in base b di un numero frazionario F:

$$F = c_{-1} \times b^{-1} + c_{-2} \times b^{-2} + c_{-3} \times b^{-3} + \ldots$$

Moltiplicando per b sia F che le **parti frazionarie** dei prodotti così ottenuti, e indicando con F_1, F_2, ... le parti frazionarie e con I_1, I_2, ... le **parti intere** di tali prodotti, si ottiene:

$$\begin{aligned}
F \times b &= \underbrace{c_{-1} \times b^0}_{I_1} + \underbrace{c_{-2} \times b^{-1} + c_{-3} \times b^{-2} + \ldots}_{F_1} \\
F_1 \times b &= \underbrace{c_{-2} \times b^0}_{I_2} + \underbrace{c_{-3} \times b^{-1} + c_{-4} \times b^{-2} + \ldots}_{F_2} \\
F_2 \times b &= \underbrace{c_{-3} \times b^0}_{I_3} + \underbrace{c_{-4} \times b^{-1} + c_{-5} \times b^{-2} + \ldots}_{F_3} \\
&\ldots
\end{aligned}$$

Ciascuna cifra c_i coincide quindi con il valore di I_i.

Operazioni in basi diverse da dieci

Valgono gli stessi principi della rappresentazione in base dieci, per esempio i meccanismi dei riporti per le addizioni e dei prestiti per le sottrazioni.

In particolare, si ricordi che in base due $1_{\text{due}} + 1_{\text{due}} = 10_{\text{due}}$.

Esempio: $1001_{\text{due}} + 11_{\text{due}}$

riporti $\rightarrow$		1	1		
	1	0	0	1	+
			1	1	=
	1	1	0	0	

Codifica binaria dei numeri

Le tecniche di codifica binaria dei numeri si basano sulla **rappresentazione** in base due, che consente di sfruttare la corrispondenza tra

- le cifre della rappresentazione in tale base (0 e 1)
- i simboli dell'alfabeto **convenzionale** della codifica binaria, $\mathcal{A} = \{0,\ 1\}$

Codifica binaria dei numeri

Le principali tecniche di codifica fanno uso di un numero n **predefinito** di bit per rappresentare il valore di qualsiasi numero. Nei calcolatori n è di norma pari al numero di bit delle celle della memoria principale.

Poiché una sequenza di n bit può assumere 2^n configurazioni diverse, la definizione di una codifica richiede due scelte principali

1. **quali valori** si devono codificare con gli n bit a disposizione?
2. **quale configurazione** degli n bit si deve associare a ciascuno di tali valori?

Codifica dei numeri naturali

Con n bit è possibile codificare non più di 2^n degli infiniti numeri naturali $\{0, 1, 2, \ldots\}$.

La codifica più intuitiva è la seguente:

1. si **sceglie** di codificare i 2^n valori dell'insieme

$$\{0,\ 1,\ 2,\ \ldots,\ 2^n - 1\}$$

2. a ciascuno di essi si associa la **configurazione** di n **bit** che corrisponde alle n **cifre** meno significative della sua **rappresentazione** in base due

Qualsiasi altro valore **non** può essere codificato.

Esempio

Per $n = 3$ si possono codificare i $2^3 = 8$ valori dell'insieme

$$\{0,\ 1,\ 2,\ \ldots,\ 2^3 - 1 = 7\}$$

La codifica è la seguente:

valore in base dieci	valore in base due (con $n = 3$ cifre)	codifica
0	000	000
1	001	001
2	010	010
3	011	011
4	100	100
5	101	101
6	110	110
7	111	111

Esercizi

1. Determinare la codifica del numero 9_{dieci} con sei bit.
2. Determinare la codifica del numero 18_{dieci} con quattro bit.
3. Determinare il valore del numero naturale la cui codifica con quattro bit sia data da 0101.

Soluzione

1. Con il procedimento delle divisioni per due si ottiene $9_{\text{dieci}} = 1001_{\text{due}}$, che con sei cifre si scrive 001001_{due}. La codifica con sei bit è quindi `001001`.
2. $18_{\text{dieci}} = 10010_{\text{due}}$; tale valore richiede più di quattro cifre, e quindi **non** può essere codificato con quattro bit. Questa conclusione si può raggiungere anche osservando che con quattro bit si possono codificare i valori da 0 a $2^4 - 1 = 15$.
3. La rappresentazione in base due coincide con la sequenza dei bit della sua codifica, ed è quindi 0101_{due}. In base dieci tale valore corrisponde a $2^2 + 2^0 = 5$.

Codifica dei numeri interi: segno e valore

Numeri interi: $\{ \ldots, -2, -1, 0, +1, +2, \ldots \}$.

La codifica in **segno e valore** si basa sul codificare **separatamente** (con bit distinti)

- il **segno**
- il **valore assoluto**

Codifica dei numeri interi: segno e valore

Usando $n > 1$ bit

- il **segno** può assumere due valori (positivo e negativo): la sua codifica richiede esattamente **un** bit
- il **valore assoluto** deve essere codificato con i restanti $n - 1$ bit, che possono assumere 2^{n-1} configurazioni distinte: come per i numeri naturali, si **sceglie** di codificare i valori

$$\{0,\ 1,\ 2,\ \ldots,\ 2^{n-1} - 1\}$$

I numeri interi codificabili sono quindi:

$$\{-2^{n-1} + 1,\ 2^{n-1} + 2,\ \ldots,\ -2,\ -1,\ 0,\ +1,\ +2,\ \ldots,\ +2^{n-1} - 1\}$$

Tutti gli altri valori **non** possono essere codificati.

Codifica dei numeri interi: segno e valore

La codifica è la seguente

- **segno**: positivo $\rightarrow$ 0, negativo $\rightarrow$ 1
- **valore assoluto**: come per i numeri naturali, si usa la **configurazione** di $n-1$ bit corrispondente alle $n-1$ cifre meno significative della **rappresentazione** in base due

Per convenzione, il bit del segno è quello più a sinistra.

Si noti che esistono **due** codifiche per lo zero.

Esempio

Per $n = 3$ si possono codificare i valori dell'insieme

$$\{-3,\ -2,\ -1,\ 0,\ +1,\ +2,\ +3\}$$

Il bit che codifica il segno è evidenziato in rosso.

valore in base dieci	valore in base due (con $n-1=2$ cifre)	codifica
-3	-11	111
-2	-10	110
-1	-01	101
-0	-00	100
$+0$	$+00$	000
$+1$	$+01$	001
$+2$	$+10$	010
$+3$	$+11$	011

Esercizi

1. Determinare la codifica in segno e valore del numero -9_{dieci}, con sei bit.
2. Determinare il numero **minimo** di bit necessari per codificare in segno e valore il numero $+18_{\text{dieci}}$.
3. Determinare il valore del numero la cui codifica in segno e valore con cinque bit sia data da `01101`.

Soluzione

1. Con il procedimento delle divisioni per due si ottiene $-9_{\text{dieci}} = -1001_{\text{due}}$, che con **cinque cifre** si scrive -01001_{due}. La codifica con **sei bit** è quindi 101001.
2. $+18_{\text{dieci}} = +10010_{\text{due}}$; questo valore è composto da **cinque** cifre significative in base due, e richiede quindi almeno **sei bit** (incluso il bit di segno) per essere codificato in segno e valore. La stessa conclusione si può raggiungere osservando che il numero maggiore in valore assoluto codificabile con $n = 6$ bit è $2^{n-1} - 1$, e quindi, per un dato numero x, il più piccolo intero n tale che $|x| \leq 2^{n-1} - 1$ è dato da $n = \lceil \log_2(|x| + 1) \rceil + 1$, dove $\lceil a \rceil$ indica il più piccolo intero non inferiore ad a; per $x = +18_{\text{dieci}}$ si ottiene $n = 6$.
3. La rappresentazione in base due del **valore assoluto** coincide con la sequenza dei **quattro** bit più a destra della codifica, ed è quindi $1101_{\text{due}} = 13_{\text{dieci}}$. Dal valore del bit di segno si deduce che il numero è positivo, e il suo valore è quindi $+13_{\text{dieci}}$.

Svantaggi della codifica in segno e valore

La codifica in segno e valore ha due svantaggi

- esistono due codifiche per lo zero: una di esse potrebbe essere usata per codificare un ulteriore numero
- le regole per eseguire operazioni aritmetiche su numeri codificati devono tener conto del segno: questo richiederebbe la realizzazione di circuiti elettronici relativamente complicati per l'esecuzione di tali operazioni sui calcolatori

Per ovviare a tali inconvenienti, nei calcolatori si usa comunemente un'altra codifica, detta **complemento a due**.

Codifica degli interi in complemento a due

Le 2^n configurazioni di n bit vengono usate per codificare

- i 2^{n-1} valori non negativi $\{0,\ +1,\ +2, \ldots,\ +2^{n-1}-1\}$
- i 2^{n-1} valori negativi $\{-2^{n-1},\ -2^{n-1}+1,\ \ldots,\ -2,\ -1\}$

Questa scelta evita la doppia codifica dello zero.

Codifica degli interi in complemento a due

La codifica è definita come segue

- a ogni numero **non negativo** $0 \leq x \leq +2^{n-1} - 1$ si assegna la stessa sequenza di n bit della codifica in segno e valore
- a ogni numero **negativo** $-2^{n-1} \leq x < 0$ viene associata la sequenza di bit corrispondente alle n cifre **meno significative** della rappresentazione in base due del numero $2^n - |x|$

Nota: il bit più a sinistra risulta essere 0 nella codifica dei numeri non negativi, mentre è 1 nella codifica dei numeri negativi. Il segno **non** è però codificato separatamente dal suo valore assoluto: non è infatti possibile ricavare il valore assoluto di un numero dai soli $n - 1$ bit più a destra della sua codifica (come si può vedere dall'esempio seguente).

Esempio

Per $n = 3$ si possono codificare i $2^3 = 8$ valori dell'insieme

$$\{-4,\ -3,\ \ldots, +2,\ +3\}$$

La codifica è la seguente:

valore in base dieci	$2^3 - \lvert x \rvert$ (solo per $x < 0$)	codifica
-4	$4_{\text{dieci}} = 100_{\text{due}}$	100
-3	$5_{\text{dieci}} = 101_{\text{due}}$	101
-2	$6_{\text{dieci}} = 110_{\text{due}}$	110
-1	$7_{\text{dieci}} = 111_{\text{due}}$	111
0		000
$+1$		001
$+2$		010
$+3$		011

Proprietà della codifica in complemento a due

Dato un numero x, $|x| \leq +2^{n-1} - 1$, codificato in complemento a due con n bit, la codifica di $-x$ si può ottenere in due passi

1. eseguendo il **complemento** della codifica di x, cioè cambiando il valore di ciascun bit $(0 \to 1,\ 1 \to 0)$
2. sommando 1 alla sequenza di bit così ottenuta, interpretandola come la rappresentazione di un numero naturale in base due

Le n cifre meno significative del numero risultante corrispondono ai bit della codifica in complemento a due di $-x$.

Questo semplice procedimento è utile per il calcolo della differenza tra numeri codificati in complemento a due.

Esempi

Con $n = 3$ bit (si veda la tabella precedente)

- codifica di -2_{dieci}: 110; codifica di $+2_{\text{dieci}}$:
  complemento: 001; somma: $001 + 1 = 010$; codifica: 010
- codifica di $+3_{\text{dieci}}$: 011; codifica di -3_{dieci}:
  complemento: 100; somma: $100 + 1 = 101$; codifica: 101
- codifica di 0: 000; codifica di -0:
  complemento: 111; somma: $111 + 1 = 1000$; codifica: 000 (si considerano le n cifre meno significative della somma)
- codifica di -4_{dieci}: 100; questo è l'unico valore di cui non si possa codificare l'opposto con $n = 3$ bit; se si applica il procedimento descritto sopra:
  complemento: 011; somma: $011 + 1 = 100$; la codifica di $+4_{\text{dieci}}$ sarebbe 100, che però è identica a quella di -4_{dieci}

Operazioni su valori codificati in complemento a due

Le operazioni di somma e sottrazione possono essere eseguite sulle sequenze di bit che codificano gli operandi, interpretando tali sequenze come numeri naturali rappresentati in base due

- la **somma** $a + b$ si esegue applicando alle codifiche di a e b il procedimento per l'addizione, indipendentemente dal loro **segno**
- la **differenza** $a - b$ si esegue come la somma $a + (-b)$, ottenendo la codifica di $-b$ con il procedimento indicato in precedenza

La codifica del risultato corrisponde alle n cifre meno significative del valore così ottenuto (le eventuali cifre successive si trascurano).

Operazioni tra valori aventi lo stesso segno possono produrre un risultato non codificabile con lo stesso numero di bit: questa situazione è detta ***overflow*** ("straripamento"), ed è identificabile dal fatto che l'n-esimo bit del risultato (a partire da destra) ha valore **diverso** dal corrispondente bit degli operandi.

Esempi: operazioni di addizione

$2 + 1$:

$+2 \rightarrow$	0	1	0	$+$
$+1 \rightarrow$	0	0	1	$=$
	0	**1**	**1**	$\rightarrow 3$

$-1 + (-3)$:

$-1 \rightarrow$		1	1	1	$+$
$-3 \rightarrow$		1	0	1	$=$
	1	**1**	**0**	**0**	$\rightarrow -4$

$2 + (-3)$:

$+2 \rightarrow$	0	1	0	$+$
$-3 \rightarrow$	1	0	1	$=$
	1	**1**	**1**	$\rightarrow -1$

$2 + 3$:

$+2 \rightarrow$	0	1	0	$+$
$+3 \rightarrow$	0	1	1	$=$
	1	**0**	**1**	*overflow*

il terzo bit da destra del risultato (in rosso) ha valore diverso rispetto a quello degli operandi: ciò è dovuto al fatto che il risultato, $+5$, non è codificabile in complemento a due con tre bit

Esempi: operazioni di sottrazione

$2 - 1 = 2 + (-1)$:

$+2 \rightarrow$		0	1	0	+
$-1 \rightarrow$		1	1	1	=
	1	**0**	**0**	**1**	$\rightarrow +1$

$-2 - 3 = -2 + (-3)$:

$-2 \rightarrow$		1	1	0	+
$-3 \rightarrow$		1	0	1	=
	1	**0**	**1**	**1**	*overflow*

(-5 non è codificabile in complemento a due con tre bit)

$1 - (-2) = 1 + 2$:

$+1 \rightarrow$	0	0	1	+
$+2 \rightarrow$	0	1	0	=
	0	**1**	**1**	$\rightarrow +3$

Esercizi

1. Determinare la codifica in complemento a due del numero -12_{dieci}, con sei bit.
2. Determinare il numero **minimo** di bit necessari per codificare in complemento a due il numero $+14_{\text{dieci}}$.
3. Determinare il valore del numero la cui codifica in complemento a due con cinque bit sia data da `11101`.

Soluzione

1. Con il procedimento delle divisioni per due si ottiene $-12_{\text{dieci}} = -1100_{\text{due}}$. Poiché il numero è negativo, la sua codifica con sei bit corrisponde alle sei cifre meno significative della rappresentazione in base due del numero $2^6 - 12 = 64 - 12 = 52_{\text{dieci}} = 110100_{\text{due}}$. La codifica è quindi 110100.

2. Il più grande numero positivo codificabile in complemento a due con n bit è $+2^{n-1} - 1$, e quindi, per un dato numero x, il più piccolo intero n tale che $|x| \leq 2^{n-1} - 1$ è dato da $n = \lceil \log_2(|x| + 1) \rceil + 1$; per $x = +14_{\text{dieci}}$ si ottiene $n = 5$.

3. Poiché il bit più a sinistra è 1, il numero codificato x è negativo. Il suo valore si ottiene con il procedimento inverso rispetto a quello usato per la codifica. Interpretando i bit della codifica come la rappresentazione in base due di un numero naturale $N = 11101_{\text{due}} = 29_{\text{dieci}}$, tale valore è per definizione pari a $2^n - |x| = 2^5 - |x|$, da cui si ottiene $|x| = 2^5 - 29 = 3_{\text{dieci}}$. Il numero codificato è quindi -3_{dieci}.

Codifica dei numeri reali in virgola fissa

La codifica in **virgola fissa** si basa sul codificare **separatamente** (con bit distinti)

- il segno, con un bit
- la parte intera, con p bit
- la parte frazionaria, con q bit

Ovviamente si deve avere $1 + p + q = n$.

La **scelta** di p e q fa parte della definizione della codifica.

Codifica dei numeri reali in virgola fissa

- Codifica del segno: positivo $\rightarrow$ 0, negativo $\rightarrow$ 1
- Codifica della parte intera: con p bit si codificano i 2^p valori $\{0,\ 1,\ 2,\ \ldots,\ 2^p - 1\}$. La codifica corrisponde alla sequenza delle p cifre **meno** significative della parte intera (espressa in base due). I numeri la cui parte intera richiede più di p cifre significative **non** possono essere codificati
- Codifica della parte frazionaria: con q bit si codificano le q cifre **più** significative della parte frazionaria (in base due). I numeri la cui parte frazionaria richiede più di q cifre significative vengono codificati approssimandola **per troncamento** (nella conversione in base due della parte frazionaria è quindi sufficiente fermarsi alle q cifre più significative)

Codifica dei numeri reali in virgola fissa

Per convenzione, il primo bit a sinistra corrisponde al segno, e i successivi alla sequenza di $p + q$ cifre della rappresentazione in base due, in ordine decrescente di potenze della base:

$$c_{p-1}\ c_{p-2}\ \ldots\ c_1\ c_0\ ,\ c_{-1}\ c_{-2}\ \ldots\ c_{-q}$$

Esempio: codifica di $-12,625_{\text{dieci}}$ con $n = 12$, $p = 5$, $q = 6$

- $-12,625_{\text{dieci}} = -1100,101_{\text{due}}$
- codifica del segno: 1
- codifica della parte intera: 01100
- codifica della parte frazionaria: 101000
- codifica dell'intero numero: 101100101000
  in dettaglio: $\underbrace{1}_{\text{segno}}\ \underbrace{01100}_{\text{p. intera}}\ \underbrace{101000}_{\text{p. frazionaria}}$

Esercizi

1. Codificare in virgola fissa il valore $+23,5078125_{\text{dieci}}$, con dodici bit, di cui cinque bit per la parte intera e sei bit per la parte frazionaria.
2. Codificare il valore $-56,25_{\text{dieci}}$, come indicato sopra.
3. Determinare il valore la cui codifica in virgola fissa, definita come sopra, sia `000101101011`.
4. Determinare una codifica in virgola fissa in grado di codificare il valore $-7,125_{\text{dieci}}$ con il **minor** numero di bit, e **senza** approssimazioni.
5. Determinare il numero più vicino a zero e quello più grande (in valore assoluto) codificabili in virgola fissa, in funzione del numero p e q di bit per la parte intera e la parte frazionaria.

Soluzione

1. $+23,5078125_{\text{dieci}} = +10111,1000001_{\text{due}}$. La parte frazionaria ha sette cifre significative, e deve essere approssimata per troncamento alle sei cifre più significative. Codifica del segno: `0`; della parte intera: `10111`; della parte frazionaria: `100000`. Codifica dell'intero numero: `010111100000`.
2. $-56,25_{\text{dieci}} = -111000,01_{\text{due}}$. La parte intera contiene sei cifre significative, mentre per la sua codifica sono disponibili solo cinque bit: questo numero **non** può essere codificato.
3. Il bit più a sinistra è `0`, quindi il numero è positivo. I successivi cinque bit corrispondono alle cifre della parte intera, che è quindi 101_{due}, mentre i restanti sei bit corrispondono alle cifre della parte frazionaria, pari a $0,101011_{\text{due}}$. Il valore codificato è quindi $+101,101011_{\text{due}} = +5,671875_{\text{dieci}}$.

Soluzione

4. $-7,125_{\text{dieci}} = -111,001_{\text{due}}$. La parte frazionaria contiene un numero finito di cifre significative (tre), e può quindi essere codificata senza approssimazioni con almeno tre bit. La parte intera contiene tre cifre significative, e deve essere codificata con almeno tre bit. La codifica richiesta si ottiene quindi con sette bit: uno per il segno, tre per la parte intera e tre per la parte frazionaria.

5. Il valore più piccolo diverso da zero che si può scrivere in base due con q cifre nella parte frazionaria è:

$$0,\underbrace{00\ldots01}_{q \text{ cifre}}$$

Tale valore corrisponde a 2^{-q}. Il valore maggiore che si può scrivere con p cifre nella parte intera e q nella parte frazionaria è:

$$\underbrace{11\ldots1}_{p \text{ cifre}},\underbrace{11\ldots1}_{q \text{ cifre}}$$

Tale valore corrisponde a $\sum_{i=-q}^{p-1} 2^i = 2^p - 2^{-q}$.

Svantaggi della codifica in virgola fissa

Per poter codificare sia valori molto grandi che valori molto piccoli in valore assoluto, è necessario un numero di bit molto elevato.

Esempio (tutti i valori sono scritti in base dieci)

- massa del sole: $\approx 1,99 \times 10^{33}$ g $\approx 2^{111}$ g
- massa dell'elettrone: $\approx 9,11 \times 10^{-28}$ g $\approx 2^{-90}$ g

Per poter codificare entrambe le grandezze (senza approssimare la massa dell'elettrone a 0 g) sono necessari almeno 111 bit per la parte intera, e almeno 90 bit per la parte frazionaria.

Notazione esponenziale

È possibile usare un numero di bit limitato codificando solo le cifre **più significative** di un valore, approssimando per troncamento quelle meno significative.

Questo è analogo alla **rappresentazione** dei numeri frazionari in **notazione esponenziale**, usata in discipline come la fisica e la chimica.

In tale notazione un valore N si scrive come $m \times b^e$, dove

- m è un numero reale
- $b > 1$ è un numero intero, di norma pari a dieci
- e è un numero intero

Esempio: $-4739,506$ può essere riscritto (in infiniti modi) come $-4739,506 \times 10^0$, $-47,39506 \times 10^2$, $-473950600 \times 10^{-5}$, ...

Notazione esponenziale

Esistono due particolari versioni della notazione esponenziale che considerano una **singola** rappresentazione per ciascun numero $N \neq 0$. In base dieci sono definite come segue

- notazione **esponenziale normalizzata**: $b = 10$, $0{,}1 \leq |m| < 1$
  - m ha parte intera nulla e prima cifra frazionaria diversa da zero
  - esempio: $-4739{,}506$ si scrive come $-0{,}4739506 \times 10^4$
- notazione **scientifica**: $b = 10$, $1 \leq |m| < 10$
  - la parte intera di m contiene una sola cifra significativa
  - in questa notazione, e viene detto **ordine di grandezza**
  - esempio: $-4739{,}506$ si scrive come $-4{,}739506 \times 10^3$

Notazione esponenziale

La notazione esponenziale può ovviamente essere usata anche in basi diverse da dieci.

Per esempio, $+1011{,}011001_{\text{due}}$ si può riscrivere in infiniti modi:
$+1011011{,}001 \times 10^{-11}$, $+10{,}11011001 \times 10^{+10}$,
$+0{,}0001011011001 \times 10^{+111}$, ...

In particolare, nella base due

- notazione esponenziale **normalizzata**:
  $b = 10_{\text{due}}$, $0{,}1_{\text{due}} \leq |m| < 1$
  esempio: $+1011{,}011001_{\text{due}} = +0{,}1011011001 \times 10^{+100}$
- notazione **scientifica**: $b = 10_{\text{due}}$, $1 \leq |m| < 10_{\text{due}}$
  esempio: $+1011{,}011001_{\text{due}} = +1{,}011011001 \times 10^{+11}$

Codifica dei numeri reali in virgola mobile

Si basa sulla rappresentazione di un numero $N \neq 0$ in base due, in notazione **scientifica**: $N = m \times b^e$, $b = 10_{\text{due}}$, $1 \leq |m| < 10_{\text{due}}$

- m è detta **mantissa**
- b è detto **base**
- e è detto **esponente**

In inglese questa codifica è detta ***floating point***.

Codifica dei numeri reali in virgola mobile

Con n bit vengono codificati separatamente (con bit distinti)

- il segno, con un bit
- le p cifre più significative della **parte frazionaria** della mantissa, con p bit, approssimando per troncamento le eventuali cifre successive diverse da zero
- l'esponente, con q bit, usando un'opportuna codifica per gli interi

Ovviamente $1 + p + q = n$; la scelta di p e q fa parte della definizione della codifica.

Si noti che con queste scelte **non** è necessario codificare

- la **parte intera** della mantissa, per definizione sempre pari a uno
- la **base**, per definizione sempre pari a due

Per convenzione, gli n bit da sinistra a destra corrispondono nell'ordine alla codifica del segno, dell'esponente e della mantissa.

Codifica dei numeri reali in virgola mobile

Esempio: codifica di $-12{,}625_{\text{dieci}}$ con $n = 12$, $p = 5$, $q = 6$, e codifica dell'esponente in **segno e valore**

- $-12{,}625_{\text{dieci}} = -1100{,}101_{\text{due}}$
- in notazione scientifica:
  $-1100{,}101_{\text{due}} = -1{,}100101_{\text{due}} \times 10_{\text{due}}^{+11_{\text{due}}}$
- codifica del segno: 1
- codifica dell'esponente (segno e valore): 000011
- codifica della mantissa: 10010 (con approssimazione per troncamento)
- codifica dell'intero numero: 100001110010
  in dettaglio: $\underbrace{1}_{\text{segno}}\ \underbrace{000011}_{\text{esponente}}\ \underbrace{10010}_{\text{mantissa}}$

Esercizi

1. Codificare in virgola mobile il valore $+23{,}5078125_{\text{dieci}}$, con dodici bit, di cui sette bit per la mantissa e quattro bit per l'esponente (codificato in segno e valore).
2. Determinare il valore la cui codifica in virgola mobile, definita come sopra, sia 000101101011.
3. Codificare il valore $-56{,}25_{\text{dieci}}$ in virgola mobile, con otto bit, di cui quattro per la mantissa e tre per l'esponente.
4. Determinare una codifica in virgola mobile in grado di codificare il valore $-17{,}125_{\text{dieci}}$ con il **minor** numero di bit, e **senza** approssimazioni.
5. Determinare il numero più piccolo e quello più grande (in valore assoluto) codificabili in virgola mobile, in funzione del numero p e q di bit della mantissa e dell'esponente.

Soluzione

1. $+23{,}5078125_{\text{dieci}} = +10111{,}1000001_{\text{due}}$
   $= +1{,}01111000001 \times 10^{+100}$. La codifica del segno è 0. La codifica dell'esponente (in segno e valore con quattro bit) è 0100. La codifica della mantissa (le sette cifre frazionarie più significative) è 0111100. La codifica di $+23,5078125_{\text{dieci}}$ è quindi 001000111100; in dettaglio: $\underbrace{0}_{\text{segno}}\ \underbrace{0100}_{\text{esponente}}\ \underbrace{0111100}_{\text{mantissa}}$.
2. Il bit di segno (quello più a sinistra) è 0, quindi il numero è positivo. La codifica dell'esponente (i successivi quattro bit) è 0010, che corrisponde a $+10_{\text{due}}$. La codifica della mantissa (i restanti sette bit) è 1101011, che corrisponde a $1,1101011_{\text{due}}$. Il valore codificato è quindi $1,1101011 \times 10^{+10}$ (in base due) $= +1,8359375 \times 2^{+2}$ (in base dieci).

Soluzione

3. $-56{,}25_{\text{dieci}} = -111000{,}01_{\text{due}} = -1{,}1100001 \times 10^{+101}$. L'esponente ($+101_{\text{due}} = +5_{\text{dieci}}$) non può essere codificato in segno e valore con tre bit, quindi il valore $-56{,}25_{\text{dieci}}$ non può essere codificato in virgola mobile usando tre bit per la codifica dell'esponente.
4. $-17{,}125_{\text{dieci}} = -10001{,}001_{\text{due}} = -1{,}0001001 \times 10^{+100}$. La codifica dell'esponente in segno e valore richiede almeno quattro bit. La mantissa contiene sette cifre frazionarie significative, per cui la sua codifica senza approssimazioni richiede almeno sette bit. Il numero minimo di bit necessari per codificare $-17{,}125_{\text{dieci}}$ senza approssimazioni è quindi dodici, con quattro bit per l'esponente e sette bit per la mantissa.

Soluzione

5. Si tratta di determinare il valore più piccolo ($x_{\min}$) e quello più grande ($x_{\max}$) che si possano scrivere in base due nella forma $m \times 2^e$, dove $1 \leq |m| < 10_{\text{due}}$ ed e è un intero, con p cifre significative nella parte frazionaria di m, e $q-1$ cifre significative per e (si ricordi che avendo a disposizione q bit, in segno e valore il valore assoluto si codifica con $q-1$ bit). È facile convincersi che il valore più piccolo si ottiene combinando il valore più piccolo di m e quello negativo e maggiore in valore assoluto di e, che con i vincoli indicati sono:

$$m = 1,\underbrace{00\ldots0}_{p \text{ cifre}}, \quad e = -\underbrace{11\ldots1}_{q-1 \text{ cifre}} = -\sum_{i=0}^{q-2} 2^i = -(2^{q-1}-1).$$

Quindi:

$$x_{\min} = 1 \times 2^{-(2^{q-1}-1)}$$

(cont.)

Soluzione

5. (cont.)

   Il valore di $x_{\max}$ si ottiene combinando il valore più grande di m e quello positivo e maggiore in valore assoluto di e

   - $m = 1,\underbrace{11\ldots1}_{p \text{ cifre}} = \sum_{i=-p}^{0} 2^i = 2^1 - 2^{-p}$,
   - $e = +\underbrace{11\ldots1}_{q-1 \text{ cifre}} = \sum_{i=0}^{q-2} 2^i = 2^{q-1} - 1$

   Quindi:

$$x_{\max} = (2 - 2^{-p}) \times 2^{2^{q-1}-1}$$

Codifica in virgola mobile: lo *standard* IEEE-754

La codifica sopra descritta presenta alcuni svantaggi. Per esempio, richiede che la mantissa abbia parte intera pari a 1 e quindi non consente di codificare senza approssimazioni il valore 0.

La codifica effettivamente usata nei calcolatori è una variante definita nel 1985 dallo Institute of Electrical and Electronic Engineers (nota come ***standard* IEEE-754**)

- prevede diversi formati
  - $n = 32$, $p = 23$, $q = 8$
  - $n = 64$, $p = 52$, $q = 11$
  - $n = 128$, $p = 112$, $q = 15$
- codifica dell'esponente: **eccesso a** 2^{q-1}
- codifica dello zero: bit di mantissa ed esponente tutti pari a 0
- alcune configurazioni degli n bit sono usate per codificare
  - il valore "*not a number*" (NaN), per es. il risultato di $0/0$
  - i valori $+\infty$ e $-\infty$
  - alcuni valori con mantissa avente parte intera nulla

Codifica dei numeri: considerazioni finali

Nei calcolatori la codifica binaria dei numeri avviene sempre con un numero **finito** di bit. Questo implica che

- non possono essere codificati numeri con valore assoluto maggiore di un certo limite
- i numeri reali vengono codificati con **precisione finita**, cioè con un numero finito di cifre frazionarie (i valori codificati sono quindi un sottoinsieme dei numeri **razionali**)

Una conseguenza importante: le operazioni aritmetiche su numeri reali possono produrre **errori di approssimazione**, i quali possono **propagarsi** e **amplificarsi** in una sequenza di calcoli.

La branca della matematica nota come **calcolo numerico** si occupa della ricerca di **algoritmi** per la risoluzione **numerica** (di norma, attraverso i calcolatori) di problemi che non siano risolubili per via analitica, come l'integrazione di equazioni differenziali, e in particolare studia gli effetti della propagazione degli errori di approssimazione.

Algoritmi: formulazione, rappresentazione, proprietà

Sommario

- Algoritmi e programmi
- Approccio alla formulazione di algoritmi e alla programmazione
- Rappresentazione di algoritmi: diagrammi di flusso
- Proprietà degli algoritmi: correttezza ed efficienza

Algoritmi e programmi

Algoritmo: descrizione del procedimento per l'esecuzione di una data operazione, espressa in modo **non ambiguo**, in un linguaggio **comprensibile** da un dato **esecutore**, in termini di una sequenza **finita** di azioni, ciascuna delle quali sia **eseguibile** dall'esecutore.

(Per una spiegazione dettagliata di questa definizione si rimanda ai **testi di riferimento**.)

Programma: rappresentazione di un algoritmo in un linguaggio "comprensibile" da un **calcolatore**.

Approccio alla programmazione

1. **Comprendere il problema** da risolvere: quali sono i **dati da elaborare** e il **risultato desiderato**?
2. Definire un procedimento risolutivo (algoritmo), espresso inizialmente anche in modo informale (per esempio in linguaggio naturale) o mediante strumenti come i diagrammi di flusso, purché **non ambiguo**
3. Se necessario, riformulare l'algoritmo in termini delle operazioni esprimibili nel linguaggio di programmazione scelto
4. Tradurre l'algoritmo in un programma codificato in tale linguaggio

Formulazione di algoritmi

Un aspetto importante consiste nella scelta delle operazioni da usare nella descrizione di un algoritmo.

Questa scelta dipende dalle operazioni eseguibili dall'esecutore (sia esso un essere umano o una macchina).

In prima battuta può essere conveniente descrivere un algoritmo in termini di operazioni di complessità arbitraria.

Se l'esecutore sarà un calcolatore, l'algoritmo dovrà essere riformulato in termini delle operazioni esprimibili nel linguaggio di programmazione scelto.

Rappresentazione di algoritmi

Un algoritmo può essere rappresentato in modo più o meno formale in diversi modi

- in linguaggio naturale
- graficamente, mediante un *diagramma di flusso*
- in uno *pseudo-codice*, ovvero in parte in linguaggio naturale e in parte usando i costrutti di un linguaggio di programmazione

Diagrammi di flusso

I **diagrammi di flusso**, o **diagrammi a blocchi**, sono uno strumento grafico per la descrizione di algoritmi.

Possono essere usati come passo intermedio verso la codifica di un algoritmo in un linguaggio di programmazione.

Una diagramma di flusso rappresenta la **sequenza** di operazioni che compongono un algoritmo

- è composto da **blocchi**, ciascuno dei quali corrisponde a una particolare operazione
- i blocchi sono collegati tra loro da frecce che indicano in modo **univoco** la **sequenza** nella quale le corrispondenti operazioni dovranno essere eseguite

Diagrammi di flusso

Per convenzione i diagrammi di flusso vengono disegnati disponendo la sequenza dei blocchi in verticale, dall'alto verso il basso.

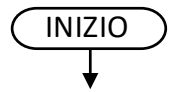
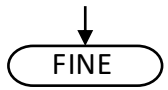
I blocchi appartengono a quattro categorie, corrispondenti a diversi tipi di operazioni

- inizio e conclusione dell'algoritmo
- "acquisizione" dei dati da elaborare e "invio all'esterno" dei risultati (ingresso/uscita – *input*/*output*, I/O)
- elaborazione (per esempio, operazioni aritmetiche)
- selezione (scelta) tra due alternative

Di seguito si descrive una delle più comuni rappresentazioni grafiche per i vari blocchi.

Blocchi di inizio e conclusione di un algoritmo

Sono rappresentati dai seguenti simboli:

- ogni diagramma di flusso deve iniziare con il blocco  e terminare con il blocco 

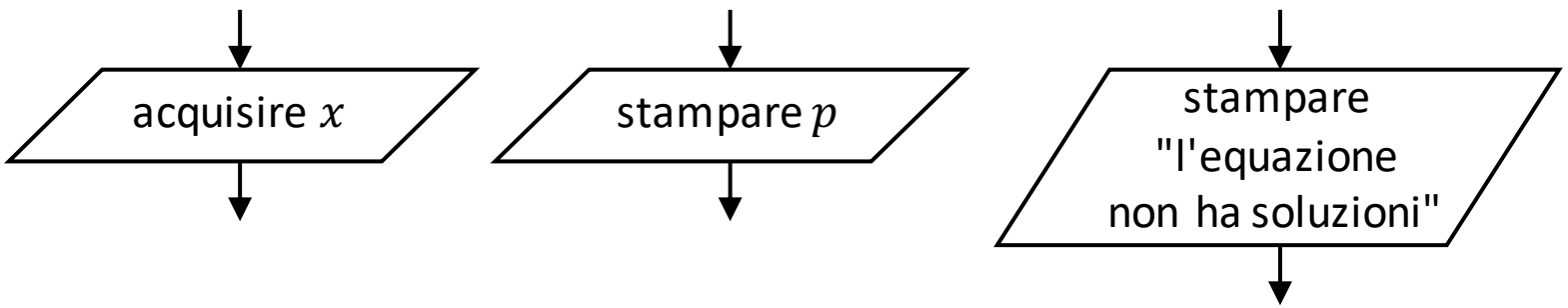
- possono esserci più occorrenze del blocco `FINE`, al termine di diverse **ramificazioni** di un diagramma di flusso originate da blocchi di *selezione*
- il blocco `INIZIO` è l'unico a non avere una freccia entrante, e il blocco `FINE` è l'unico a non averne una uscente

Blocchi di ingresso/uscita

Sono rappresentati da parallelogrammi, e rappresentano le operazioni di

- "acquisizione" (per esempio, mediante la tastiera) dei valori da elaborare (i valori "d'ingresso" dell'algoritmo) e la loro associazione a identificatori simbolici: di norma si usa il termine `acquisire` seguito dal nome dell'identificatore
- "stampa" (per esempio, sullo schermo) di messaggi o dei risultati dell'algoritmo: di norma si usa il termine `stampare` seguito da un'espressione o da un messaggio (scritto tra doppi apici)

Alcuni esempi:



Blocchi di ingresso/uscita

Durante l'esecuzione di un programma da parte di un calcolatore

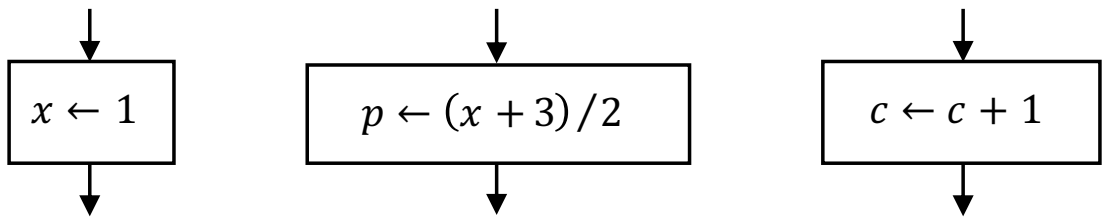
- l'associazione di un valore a un identificatore simbolico corrisponde alla sua scrittura in una cella di memoria
- le operazioni d'ingresso /uscita avvengono attraverso opportune periferiche d'ingresso: tastiera, memoria secondaria, monitor, stampanti, ecc.

Blocco di elaborazione

È rappresentato da un rettangolo che contiene un'**espressione** (di norma, aritmetica) il cui valore dovrà essere associato a un **identificatore simbolico**, scritta con la seguente sintassi:

$$\texttt{identificatore} \leftarrow \texttt{espressione}$$

Esempi:



In un calcolatore ciò corrisponde alla scrittura in una cella di memoria di un valore esplicito o del risultato di un calcolo, e costituisce una delle operazioni fondamentali esprimibili nel linguaggio macchina e in tutti i linguaggi di programmazione.

Blocco di elaborazione

L'operazione indicata in un blocco di elaborazione viene eseguita in due fasi

1. si calcola il valore dell'espressione a destra del simbolo $\leftarrow$
2. si associa tale valore all'identificatore

Un'espressione che contiene identificatori, come $(x + 3)/2$, ha senso solo se a ciascuno di essi è già stato associato un valore.

In particolare, un'operazione come $c \leftarrow c + 1$ corrisponde a incrementare di un'unità il valore associato all'identificatore c.

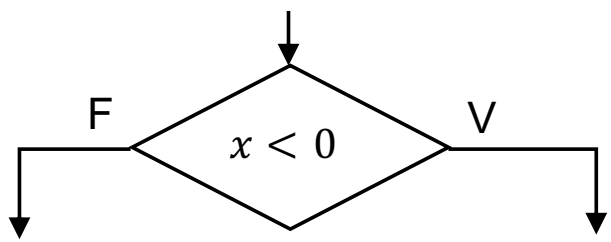
Blocco di selezione

È rappresentato da un rombo con *due* frecce uscenti.

Esprime la scelta tra due sequenze **alternative** di operazioni, una sola delle quali verrà eseguita, in base al verificarsi o meno di una data condizione.

La condizione consiste di norma in un confronto tra due **espressioni** per mezzo degli operatori $<$, $\leq$, $=$, $\neq$, $\geq$, $>$.

Le frecce uscenti dal blocco di selezione sono associate a "etichette" indicanti il verificarsi (Vero) o meno (Falso) della condizione. Un esempio:



Formulazione di algoritmi

Quando si formula un algoritmo, o si vuole verificare la correttezza o comprendere il funzionamento di un algoritmo dato, può essere utile eseguirlo (per esempio, usando carta e matita) per alcuni possibili valori dei dati d'ingresso.

A questo scopo è conveniente tener traccia del valore **corrente** associato a ciascuno degli identificatori simbolici dopo ogni operazione di acquisizione o di elaborazione.

In particolare, se all'identificatore coinvolto era già associato un valore, questo sarà **sostituito** dal nuovo valore.

Quando l'algoritmo sarà tradotto in un programma ed eseguito da un calcolatore, ciascun identificatore corrisponderà a una cella di memoria nella quale verranno memorizzati i valori corrispondenti.

Esempi

Di seguito si riportano esempi di formulazione e rappresentazione di algoritmi mediante diagrammi di flusso. Per comprenderne il funzionamento si suggerisce di procedere come indicato sopra.

Per analogia con i linguaggi di programmazione si useranno nei blocchi di elaborazione solo operazioni aritmetiche composte da

- operatori di somma ($+$), sottrazione ($-$), moltiplicazione ($\cdot$), divisione ($/$) e modulo (`mod`, resto di una divisione tra interi)
- operandi che consistono in numeri o identificatori
- parentesi tonde per indicare la precedenza tra gli operatori

Inoltre nei blocchi di selezione si useranno solo operazioni di confronto mediante gli operatori $<$, $\leq$, $=$, $\neq$, $\geq$, $>$.

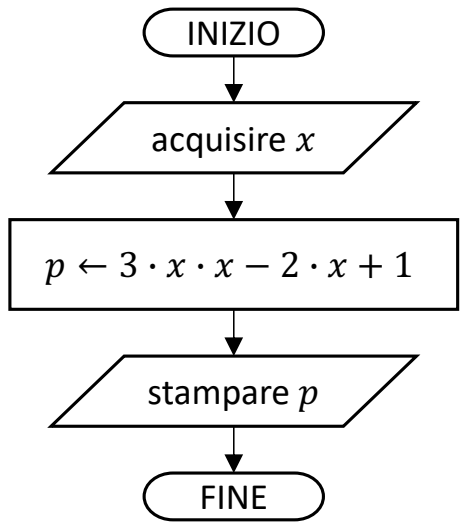
Questo corrisponde ad assumere che l'esecutore sia in grado di eseguire solo tali operazioni.

Esempio 1: calcolo di un polinomio

Calcolare il valore del polinomio $3x^2 - 2x + 1$ *per un dato valore di* x*, che dovrà essere acquisito durante l'esecuzione dell'algoritmo*

In altre parole, l'algoritmo dovrà essere in grado di calcolare il valore di tale polinomio per **qualsiasi** valore di x, non per uno specifico valore che sia già noto a chi formula l'algoritmo.

L'algoritmo può essere descritto da una semplice sequenza di blocchi di ingresso/uscita e di elaborazione.



Esempio 2: valore assoluto

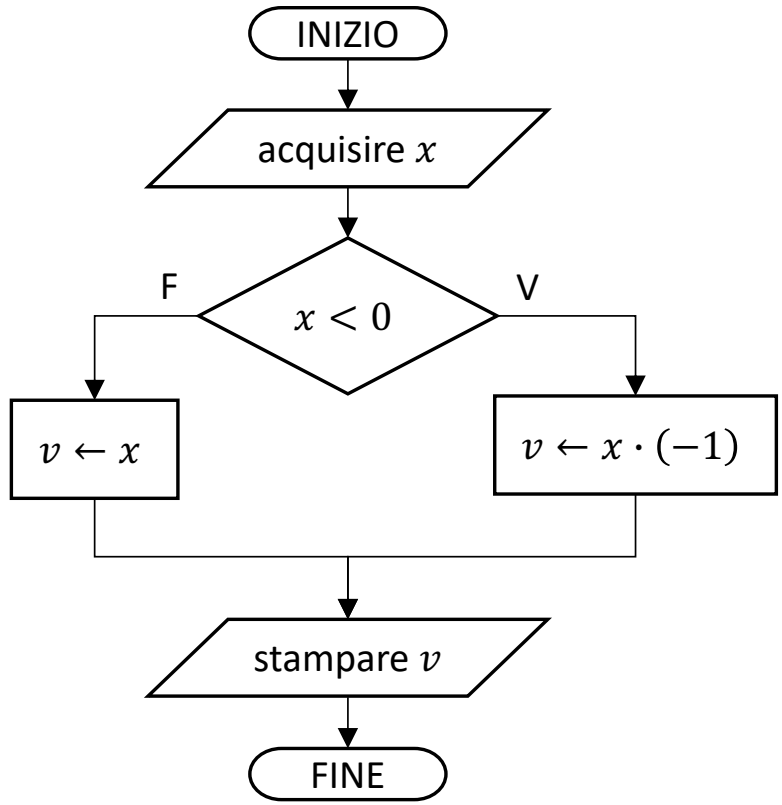
Calcolare il valore assoluto di un dato numero

Anche in questo caso s'intende che il dato da elaborare dovrà essere acquisito durante l'esecuzione dell'algoritmo.

Avendo a disposizione le sole operazioni indicate in precedenza non è difficile concludere che il calcolo del valore assoluto richiede l'uso del **blocco di selezione**.

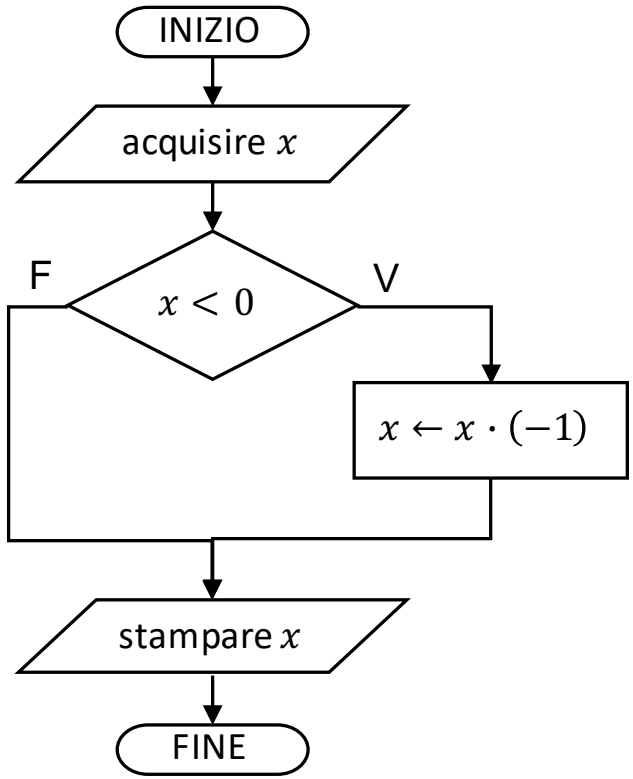
Esempio 2: valore assoluto

Una possibile soluzione fa uso di un identificatore (una cella di memoria), di nome v, per memorizzare il risultato.



Esempio 2: valore assoluto

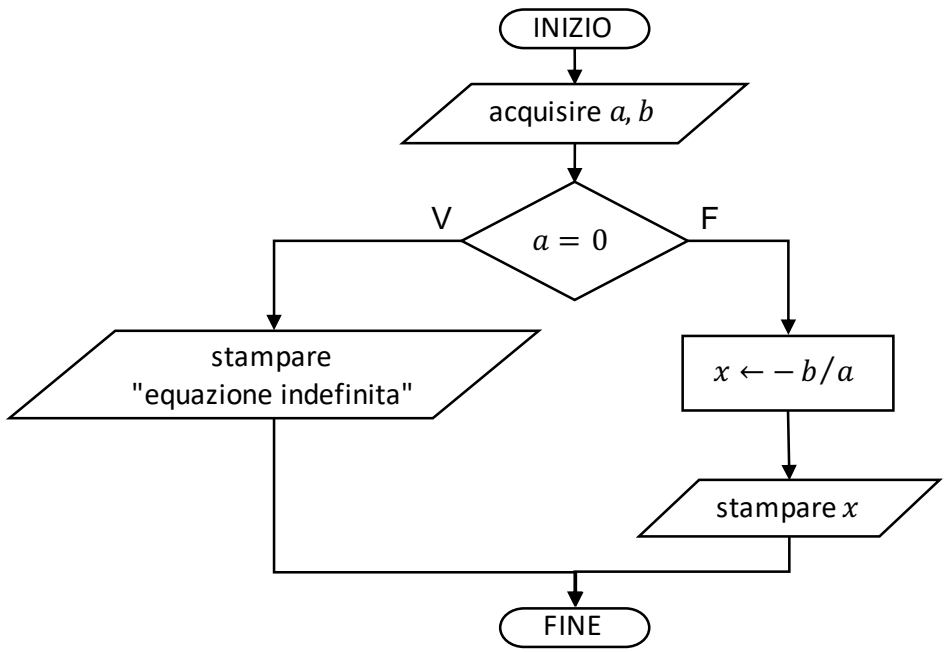
Una soluzione alternativa: si associa il risultato allo stesso identificatore (x) che contiene inizialmente il valore da elaborare. Se tale valore è positivo non è necessario svolgere nessuna elaborazione, dato che il risultato coincide con il valore stesso. Pertanto la ramificazione del blocco di selezione corrispondente al caso in cui $x < 0$ è falso non prevede l'esecuzione di nessuna operazione.



Esempio 3: equazioni di primo grado

Calcolare la radice di un'equazione di primo grado, $ax + b = 0$, *per valori dati dei coefficienti*

Prima di valutare l'espressione $-b/a$ è necessario verificare che il coefficiente a sia diverso da zero, attraverso un blocco di selezione.



Esempio 4: equazioni di secondo grado

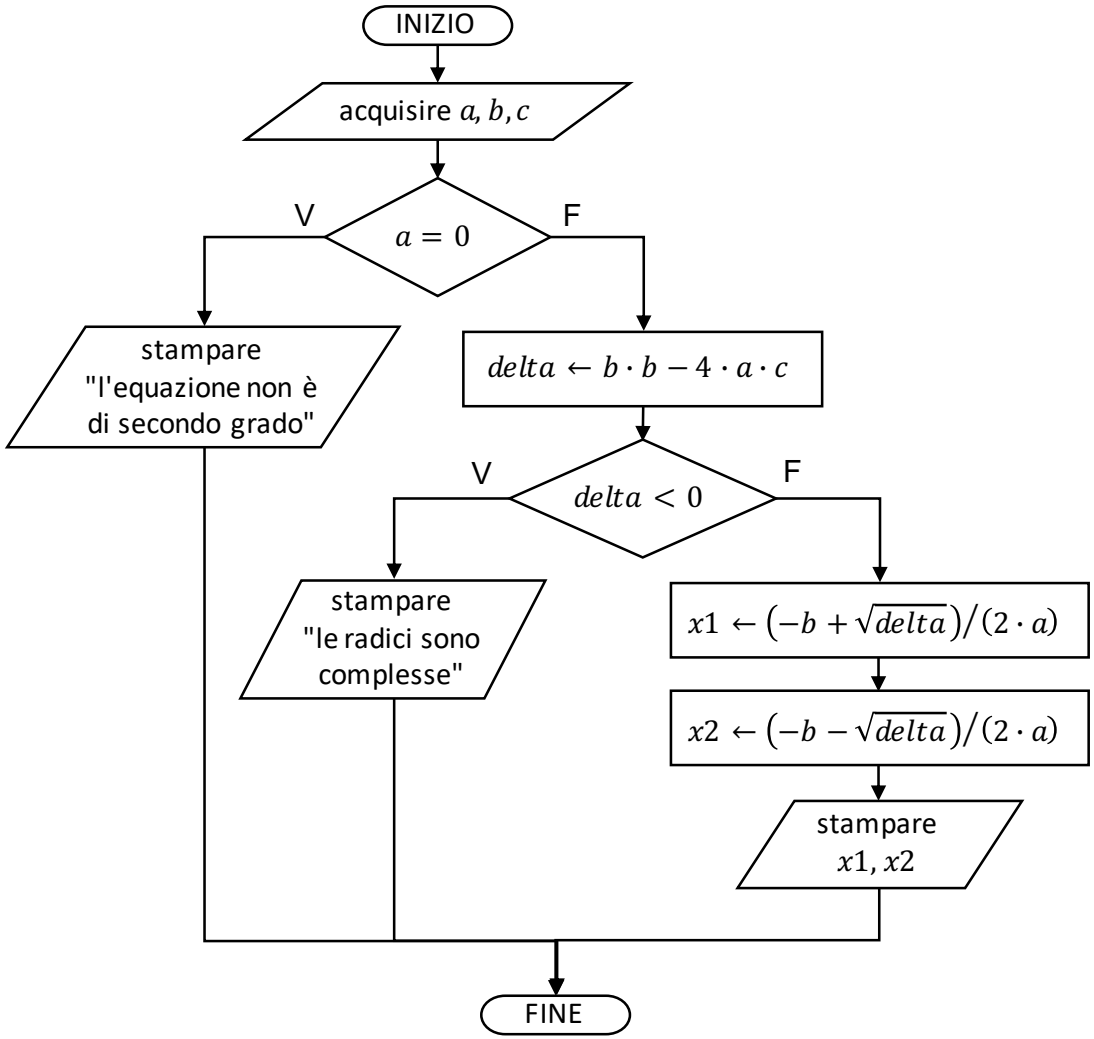
Calcolare le radici di un'equazione di secondo grado:

$$ax^2 + bx + c = 0$$

per valori dati dei coefficienti. Si assuma che l'esecutore sia in grado di calcolare radici quadrate, ma solo di valori non negativi. Se le radici fossero complesse, l'algoritmo dovrebbe prevedere solo la stampa di un messaggio opportuno

Anche in questo caso è necessario verificare che il coefficiente a sia diverso da zero. In tal caso si deve anche verificare che le radici siano reali, cioè che il termine $b^2 - 4ac$ sia non negativo.

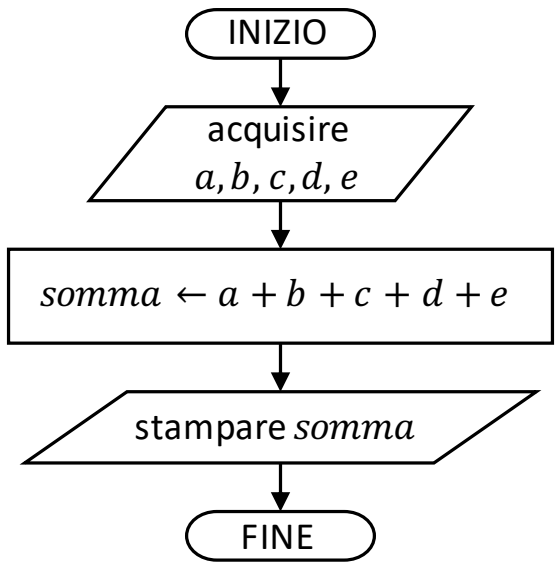
Esempio 4: equazioni di secondo grado



Esempio 5: sommatoria

Calcolare la somma di una data sequenza di cinque numeri

Una semplice soluzione consiste nell'associare gli addendi a cinque identificatori distinti, e nell'ottenerne la somma con un'unica espressione:



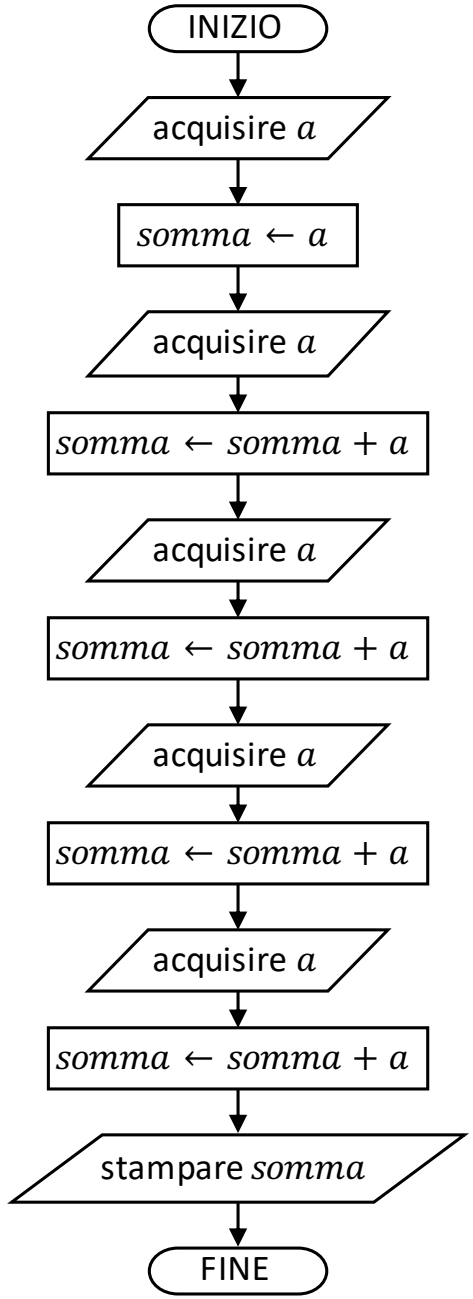
Esempio 5: sommatoria

La soluzione precedente è però poco pratica nel caso in cui il numero dei valori da sommare sia elevato, poiché bisognerebbe introdurre un numero corrispondente di identificatori, scrivendone i nomi sia nel blocco di acquisizione che in quello di elaborazione.

Una soluzione alternativa, e più aderente al procedimento di calcolo seguito da un esecutore umano, consiste nell'acquisire un valore alla volta, addizionandolo alla somma dei valori acquisiti in precedenza.

Questo consente di usare solo due identificatori, in quanto richiede di "ricordare" in ogni istante solo l'ultima somma parziale e il nuovo valore da sommare a essa (come caso particolare, la prima somma parziale corrisponde al primo addendo)

Esempio 5: sommatoria



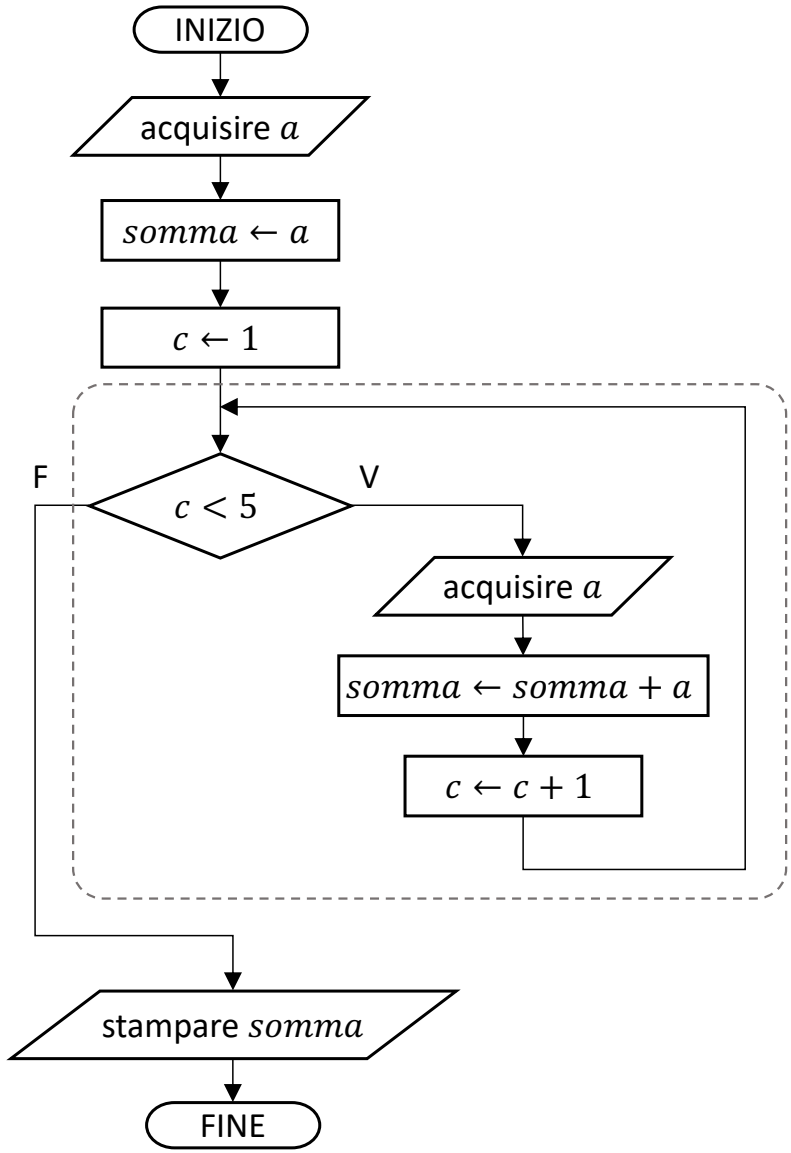
Esempio 5: sommatoria

Anche la soluzione precedente ha tuttavia uno svantaggio: richiede di tracciare più volte (tante quante sono gli addendi) la sequenza di due blocchi corrispondente all'acquisizione di un nuovo valore e all'esecuzione di una somma.

Una soluzione migliore, anche se non banale, è mostrata nella pagina seguente. Essa consiste nell'usare uno **schema iterativo** (evidenziato dal riquadro tratteggiato) che esprime in modo conciso la ripetizione (iterazione) di una *stessa* sequenza di blocchi per un certo numero di volte. A questo scopo si usa un identificatore (di nome c) per contare il numero di ripetizioni svolte.

Si noti che dopo ogni iterazione (acquisizione di un nuovo valore ed esecuzione di una somma) l'esecuzione riprende dal blocco di selezione.

Esempio 5: sommatoria



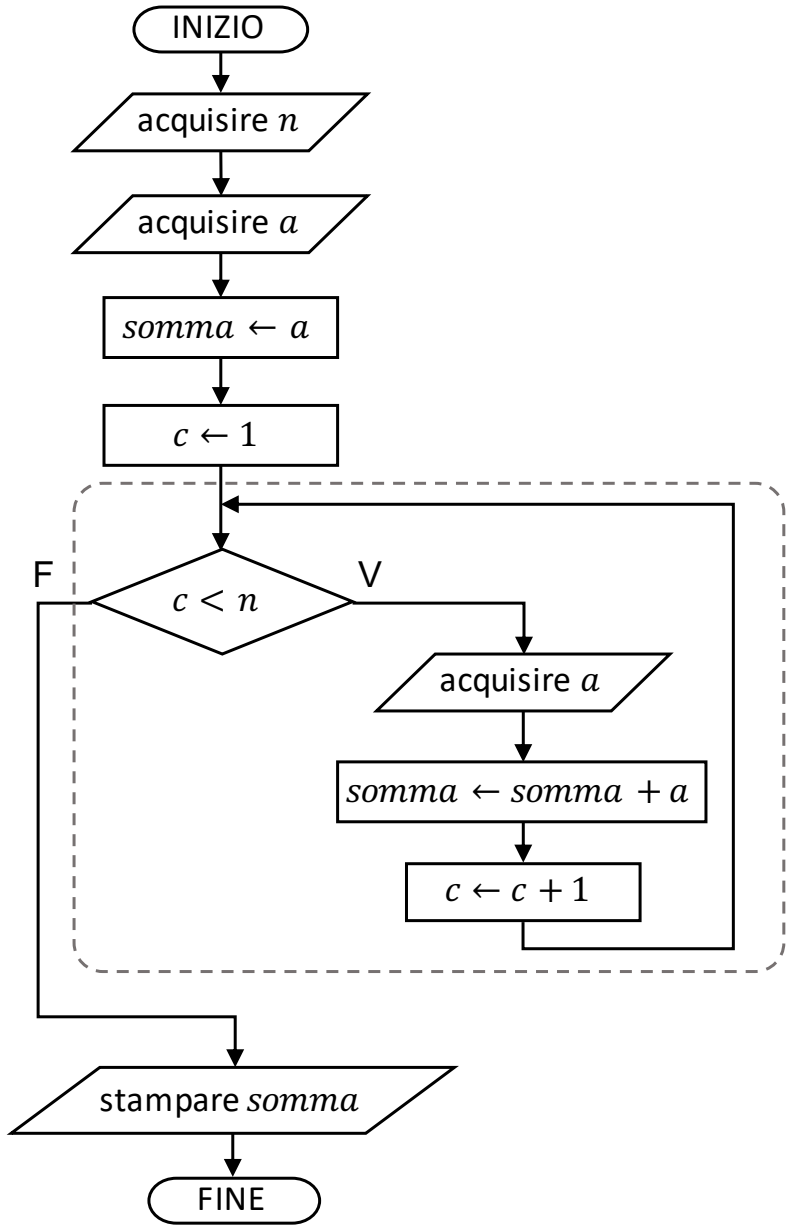
Esempio 6: sommatoria

Calcolare la somma di una sequenza di numeri di una data lunghezza (s'intende che la lunghezza della sequenza sarà nota solo al momento dell'esecuzione: l'algoritmo deve quindi essere in grado di elaborare sequenze di lunghezza qualsiasi)

Poiché la lunghezza della sequenza non è nota nel momento in cui si formula l'algoritmo, è **necessario** usare uno schema iterativo.

Nella soluzione mostrata di seguito, il numero di addendi viene acquisito come primo dato d'ingresso, e viene poi usato per la verifica del numero d'iterazioni svolte.

Esempio 6: sommatoria



Esempio 7: fattoriale

Calcolare il fattoriale n! di un dato intero non negativo n, definito come segue:

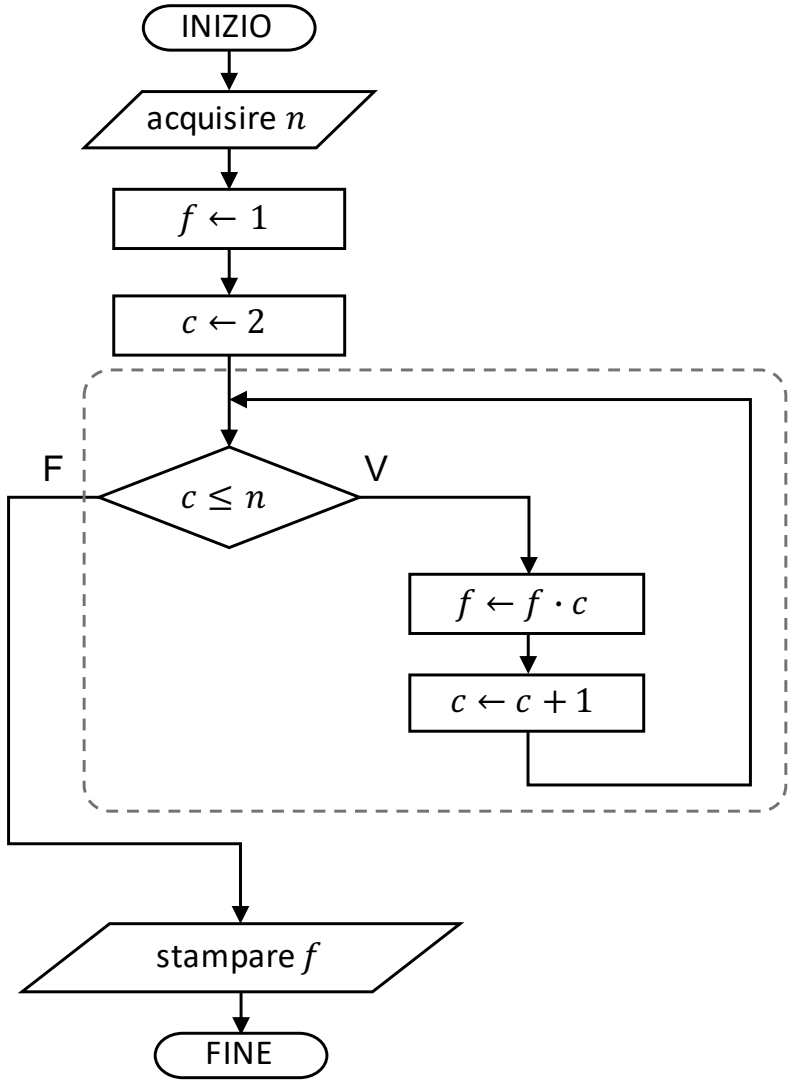
$$\begin{aligned} n! &= 1 \times 2 \times 3 \times \ldots \times (n-1) \times n, \text{ se } n > 0 \\ 0! &= 1 \end{aligned}$$

Per semplicità si può assumere che il valore acquisito sia un numero intero non negativo, evitando di eseguire la verifica

Anche in questo caso è necessario uno schema iterativo (si veda la pagina seguente). I valori da moltiplicare vengono facilmente calcolati (e associati all'identificatore c) in funzione del valore di n.

Si noti che l'assegnamento iniziale $f \leftarrow 1$ corrisponde al fattoriale dei primi due numeri naturali (0! e 1!): per questo motivo, se $n > 1$ è sufficiente proseguire il calcolo a partire dal terzo fattore, cioè 2.

Esempio 7: fattoriale



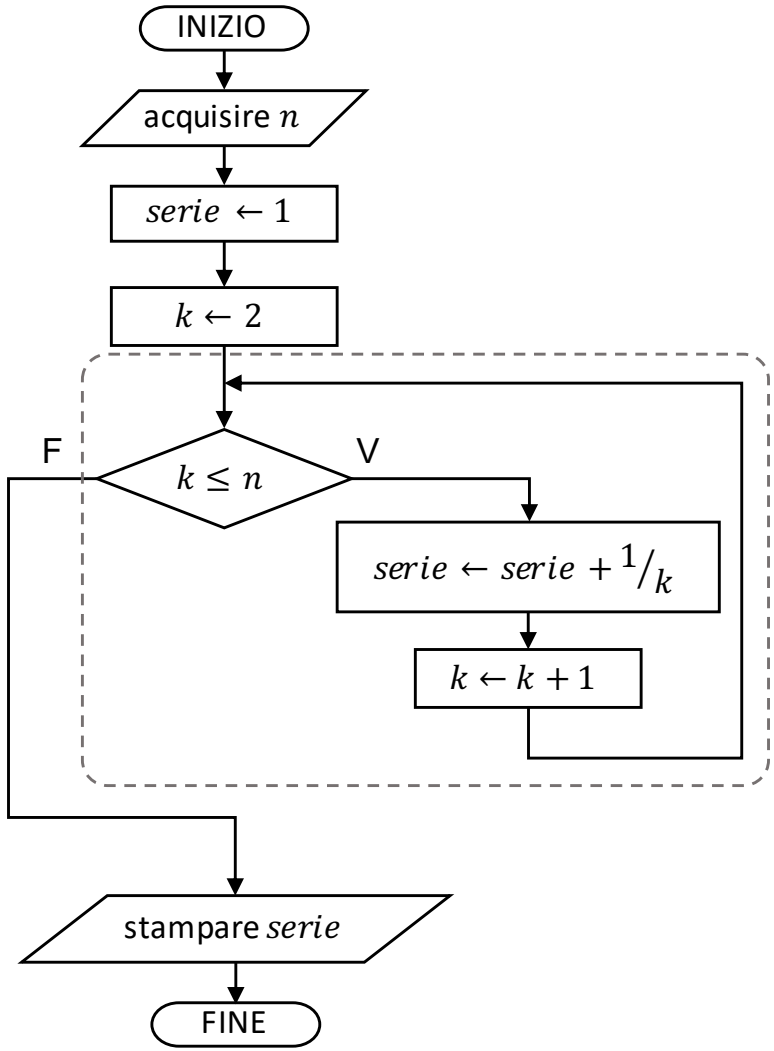
Esempio 8: serie armonica

Calcolare la somma dei primi n termini della serie armonica, per un dato valore di n:

$$\sum_{k=1}^{n} \frac{1}{k}$$

Si può formulare un algoritmo molto simile ai due precedenti. Si noti che all'identificatore *serie* viene associato inizialmente il valore 1, che corrisponde al primo termine della serie (per $k = 1$): per questo motivo il valore iniziale dell'identificatore k è 2, corrispondente al secondo termine della serie.

Esempio 8: serie armonica

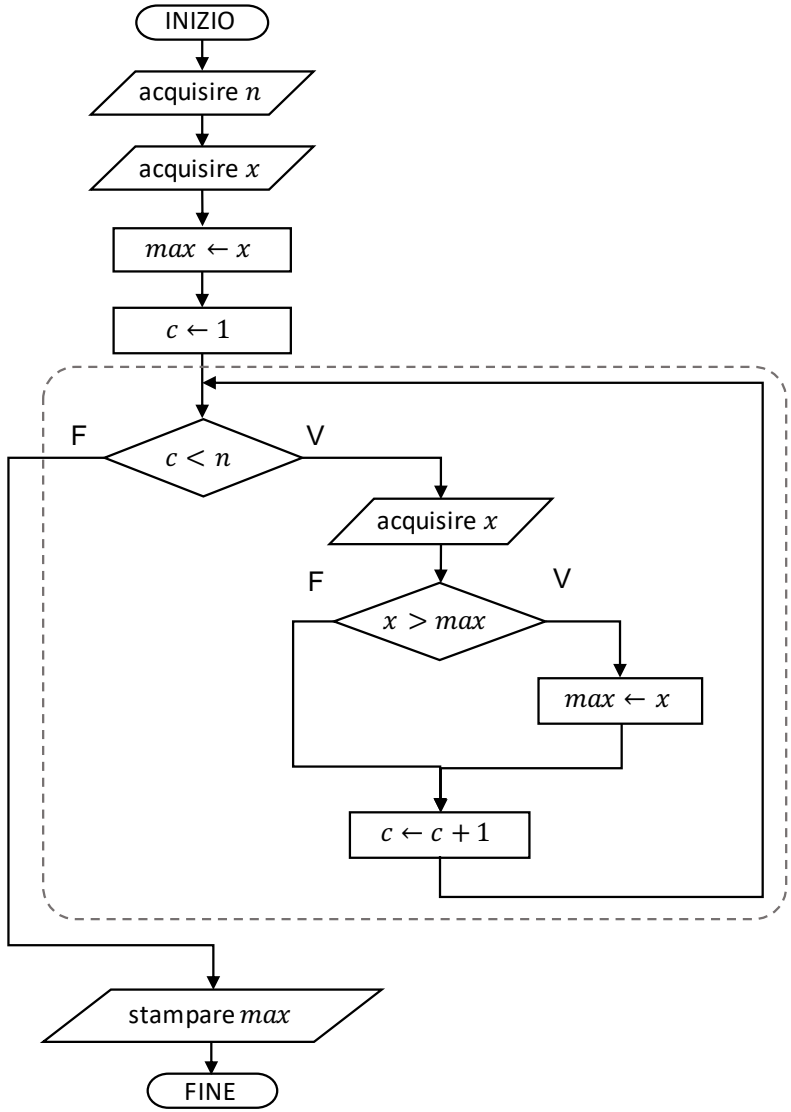


Esempio 9: massimo di un insieme di numeri

Calcolare il valore più grande di una sequenza di numeri di una data lunghezza

L'algoritmo proposto è basato su una logica simile a quella degli algoritmi che calcolano una sequenza di somme o di prodotti: anche in questo caso è sufficiente acquisire i valori da elaborare uno alla volta, tenendo traccia solo del valore più grande tra quelli già acquisiti e del valore successivo.

Esempio 9: massimo di un insieme di numeri



Esempio 9: massimo di un insieme di numeri

I valori associati agli identificatori del precedente diagramma di flusso sono i seguenti:

- *max*: il valore più grande tra quelli **già acquisiti**
- *x*: il valore **successivo** della sequenza
- *n*: la lunghezza della sequenza
- *c*: il numero di elementi **già acquisiti**

Secondo la logica sopra descritta il valore iniziale di *max* dovrà essere pari al primo elemento della sequenza; gli eventuali elementi successivi (se $n > 1$) vengono acquisiti attraverso uno schema iterativo, all'interno del quale il valore associato a *max* verrà aggiornato ogni qual volta il nuovo elemento acquisito fosse maggiore del valore attuale di *max*.

Esempio 10: massimo comun divisore

Calcolare il massimo comun divisore (MCD) di due numeri naturali dati

Un procedimento (algoritmo) ben noto si basa sulla scomposizione dei due numeri m e n in fattori primi: il loro MCD è dato dal prodotto dei fattori comuni, ciascuno elevato a una potenza pari all'esponente più piccolo con il quale esso compare nelle due scomposizioni.

Tale algoritmo non è però facilmente rappresentabile per mezzo delle operazioni elementari considerate finora (si ricorda che tali operazioni corrispondono in buona misura a quelle esprimibili nei linguaggi di programmazione di alto livello).

Esempio 10: massimo comun divisore

Un algoritmo più semplice da esprimere si può formulare partendo dalla definizione del MCD di due numeri m e n, ovvero: il più grande intero che sia divisore di entrambi.

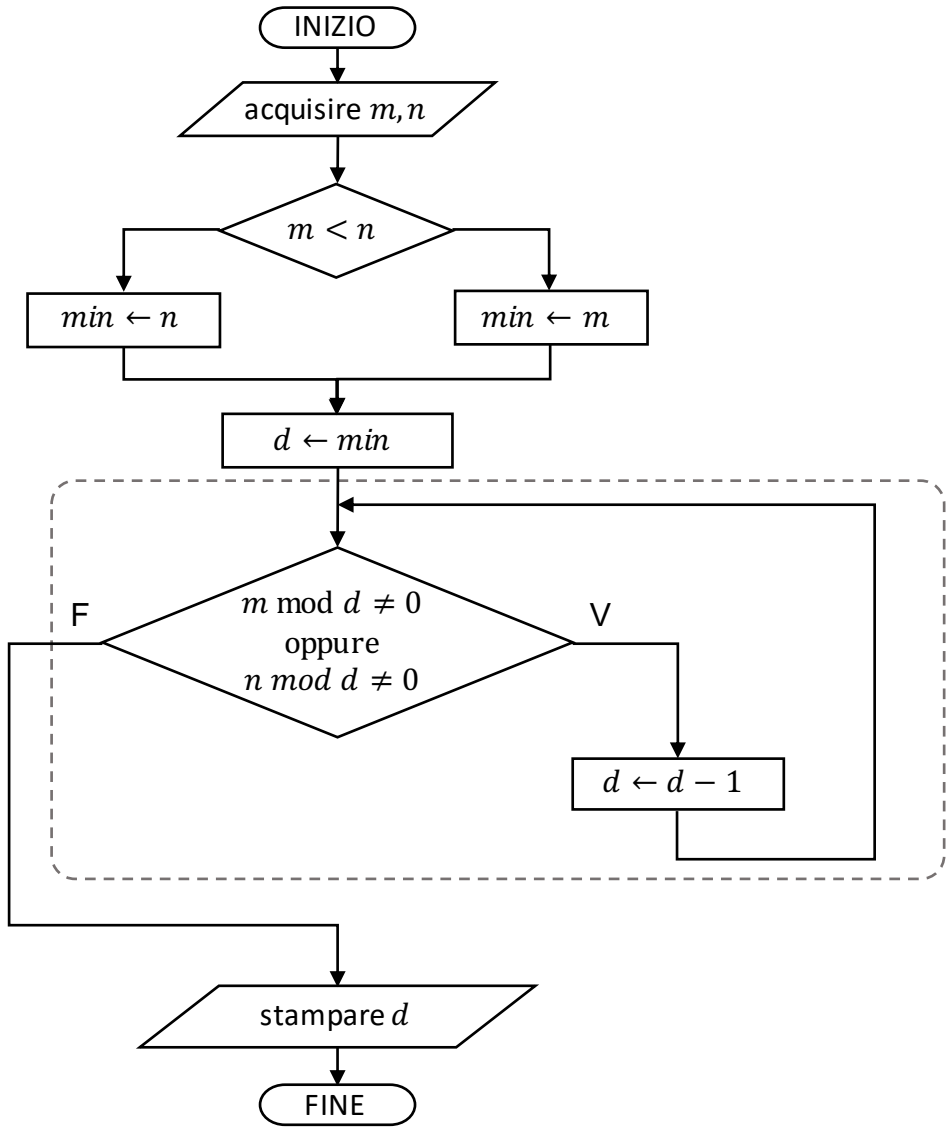
Si osservi poi che il MCD è sempre compreso tra 1 e $\min\{m, n\}$

- $\text{MCD}(m, n) = 1$, se m e n sono primi tra loro
- $\text{MCD}(m, n) = \min\{m, n\}$, se $\min\{m, n\}$ è divisore di $\max\{m, n\}$

Segue facilmente che il MCD può essere calcolato considerando a ritroso i valori da $\min\{m, n\}$ a 1, e arrestandosi non appena si trovi un divisore comune: per definizione questo sarà pari a $\text{MCD}(m, n)$.

Si noti che per verificare se un numero è divisore di un altro si può usare l'operatore aritmetico *modulo*.

Esempio 10: massimo comun divisore



Esempio 10: massimo comun divisore

Nell'algoritmo precedente si determina prima di tutto il più piccolo tra i due numeri, associandone il valore all'identificatore *min*.

Successivamente si considerano a ritroso i valori da *min* a 1 attraverso uno schema iterativo, associandoli all'identificatore *d*.

In ogni passo dell'iterazione il blocco di selezione verifica se il valore attuale di *d* sia un divisore comune di *m* e *n*

- se *d* **non** è un divisore comune, il suo valore viene decrementato di una unità e l'iterazione riprende analizzando il **nuovo** valore
- se *d* **è** un divisore comune, il suo valore è per definizione il MCD tra *m* e *n*: in questo caso l'algoritmo termina stampando tale valore

Esercizio: minimo comune multiplo

Calcolare il minimo comune multiplo (MCM) di due numeri naturali dati

La soluzione viene lasciata come esercizio.

Si suggerisce di procedere in modo analogo all'esempio del calcolo del MCD.

Il teorema di Böhm-Jacopini

I diagrammi di flusso visti finora usano tre schemi di esecuzione

- esecuzione sequenziale
- selezione
- iterazione

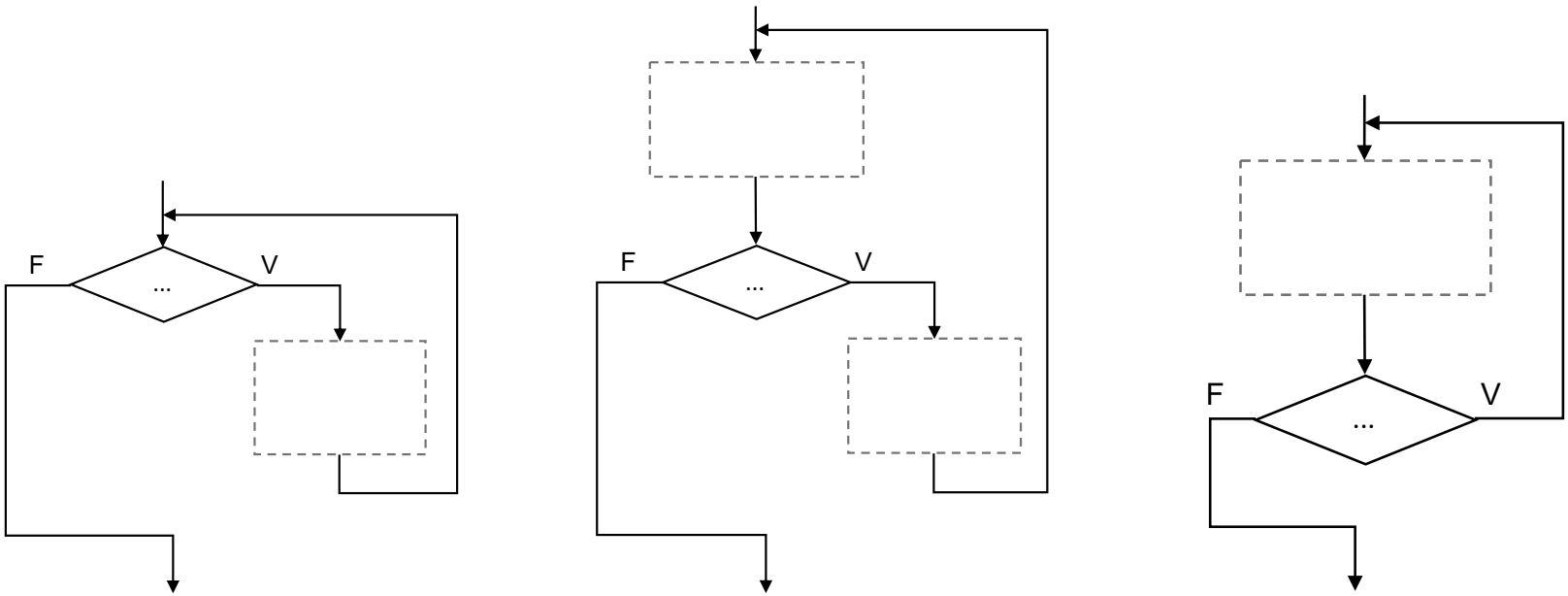
Uno dei risultati teorici fondamentali nel campo dell'informatica (noto come teorema di Böhm-Jacopini) afferma che attraverso tali schemi è possibile esprimere **qualsiasi** algoritmo.

Per questo motivo essi sono alla base di molti linguaggi di programmazione. In particolare, linguaggi di alto livello come Python esprimono tali schemi mediante

- l'**istruzione condizionale** (selezione)
- l'**istruzione iterativa**
- l'**ordine** nel quale le istruzioni del programma sono scritte

Osservazione sugli schemi iterativi

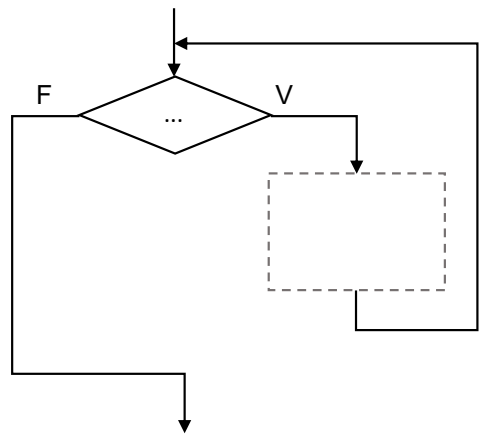
Nei diagrammi di flusso gli schemi iterativi possono avere diverse strutture, come per esempio quelle mostrate in basso, nelle quali i riquadri tratteggiati indicano una **qualsiasi** sequenza di blocchi.



Si può dimostrare che ogni schema iterativo può essere ricondotto a uno schema equivalente avente la struttura a sinistra oppure quella a destra.

Osservazione sugli schemi iterativi

Nei linguaggi di programmazione che includono l'**istruzione iterativa**, questa è sempre presente in una forma che corrisponde allo schema mostrato in basso, nel quale la prima operazione consiste in un blocco di selezione.



Per questo motivo nei diagrammi di flusso è conveniente rappresentare gli schemi iterativi usando tale struttura: ciò consente di tradurre immediatamente tali schemi in istruzioni iterative, in qualsiasi linguaggio.

Formulazione di algoritmi: problemi e istanze

Si osservi che negli esempi precedenti si richiede non la risoluzione di una specifica **istanza** di un problema (per esempio, sommare i numeri $5,\ -7,\ 4,\ 9,\ -2$), ma la definizione di un procedimento in grado di risolvere **tutte** le possibili istanze di tale problema (per esempio, sommare una **qualsiasi** sequenza di cinque numeri).

Più precisamente, per **istanza** s'intende uno specifico valore dei dati d'ingresso di un problema (per esempio, la sequenza $5,\ -7,\ 4,\ 9,\ -2$, nel caso della somma di cinque numeri).

Di norma gli algoritmi di interesse sono quelli che consentono di risolvere tutte le possibili istanze di un problema (o almeno un loro specifico sottoinsieme).

Osservazioni finali

Alcune osservazioni generali sulla formulazione di algoritmi, che possono essere ricavate dagli esempi precedenti

- possono esistere **diversi** algoritmi in grado di risolvere uno **stesso** problema
- algoritmi diversi per risolvere uno stesso problema possono essere più o meno facili da formulare e da descrivere (si veda il caso del MCD), e possono richiedere l'esecuzione di un numero diverso di operazioni
- in alcuni casi problemi apparentemente diversi possono essere risolti con algoritmi simili (si vedano gli algoritmi per la somma di una sequenza di numeri e per il calcolo del MCD)

Proprietà degli algoritmi: correttezza

Un algoritmo è **corretto** se produce il risultato desiderato per ogni possibile istanza del problema.

Il metodo apparentemente più diretto per verificare la correttezza di un algoritmo consiste nell'eseguirlo per **tutte** le istanze del problema. Tuttavia nella pratica questo è spesso impossibile

- il numero d'istanze può essere molto grande (in teoria anche infinito, come nel caso del MCD tra due numeri naturali)
- un algoritmo **non** corretto potrebbe **non** terminare mai per qualche istanza del problema

Di norma la correttezza deve quindi essere dimostrata con procedimenti analoghi alle dimostrazioni dei teoremi matematici.

La **non** correttezza può invece essere dimostrata individuando anche un singolo **controesempio**, cioè un'istanza per la quale l'algoritmo **non** produce il risultato desiderato.

Proprietà degli algoritmi: efficienza

Un algoritmo $\mathcal{A}$ è più **efficiente** di un algoritmo $\mathcal{B}$, se risolve lo stesso problema usando una quantità inferiore di **risorse**. Questa proprietà è anche indicata con il termine **complessità computazionale**.

Nel caso in cui l'esecutore sia un calcolatore le risorse necessarie sono di due tipi

- quantità di memoria (complessità **spaziale**)
- tempo d'esecuzione (complessità **temporale**)

Il tempo d'esecuzione dipende però dalle caratteristiche del calcolatore (CPU, capacità di memoria, ecc.). Per valutare la complessità temporale indipendentemente dallo specifico calcolatore si considera perciò il numero di **operazioni** richieste dall'algoritmo.

La valutazione dell'efficienza viene di norma svolta nel **caso peggiore**, cioè considerando l'istanza che richiede la maggior quantità di memoria o il numero maggiore di operazioni.

Linguaggi e ambienti di programmazione

Il linguaggio macchina

I calcolatori sono in grado di "comprendere" ed eseguire solo algoritmi codificati in linguaggio macchina (**programmi**).

Il linguaggio macchina è detto di **basso livello**, poiché le sue istruzioni corrispondono a operazioni molto elementari e fanno riferimento ai dettagli dell'organizzazione *hardware* di un calcolatore (indirizzi delle celle di memoria, registri della CPU, ecc.).

Le istruzioni dei programmi in linguaggio macchina devono essere a loro volta rappresentate in codifica binaria. I programmatori usano per comodità una rappresentazione simbolica equivalente (composta da simboli quali `ADD`, `LOAD`, `JUMP`), detta linguaggio **Assembly**.

Il linguaggio macchina

Il linguaggio macchina è composto da un numero molto limitato di istruzioni. Questo rende la programmazione in tale linguaggio un'attività piuttosto complessa.

Le istruzioni del linguaggio macchina possono essere suddivise in quattro categorie

- **elaborazione**: operazioni logico-aritmetiche elementari tra due operandi (per es.: addizione, sottrazione, moltiplicazione, divisione, confronto con zero)
- **memorizzazione**: trasferimento di un dato (il contenuto di una cella di memoria) tra CPU e memoria principale
- **trasferimento**: trasferimento di un dato tra due registri della CPU
- **controllo**: scelta della successiva istruzione da eseguire in funzione del verificarsi o meno di una data condizione

Il linguaggio macchina

Per semplificare l'attività di programmazione, a partire dagli anni '50 sono stati introdotti diversi linguaggi di **alto livello**, che consentono di rappresentare gli algoritmi in una forma più vicina al linguaggio naturale rispetto al linguaggio macchina.

I linguaggi di alto livello possiedono in particolare le seguenti caratteristiche che li differenziano dal linguaggio macchina

- non fanno esplicito riferimento agli indirizzi delle celle di memoria, ma usano per esse nomi simbolici (detti **variabili**)
- indicano le istruzioni con termini del linguaggio naturale (per esempio, `if`, `while`, `for`)
- consentono di rappresentare espressioni aritmetiche composte da più operazioni elementari, con una notazione simile a quella usata in matematica
- consentono di rappresentare dati composti da aggregazioni di valori più elementari (per esempio, sequenze di caratteri o di numeri)

Esempi di linguaggi di alto livello

- Fortran (*FORmula TRANslator*)
- Lisp (*LISt Processor*)
- Cobol (*COmmon Business Oriented Language*)
- C
- Basic
- Pascal
- SmallTalk
- C++
- Java
- **Python**
- Matlab
- ...

Linguaggi di alto livello

Tutti i linguaggio di alto livello hanno la stessa **capacità espressiva**, cioè consentono di rappresentare **qualsiasi** algoritmo.

Ciascun linguaggio è stato però ideato con lo scopo di rappresentare più facilmente alcuni tipi di algoritmi, per esempio

- FORTRAN: calcolo numerico
- Lisp: elaborazione di strutture dati complesse (liste, grafi, ecc.), tipiche delle applicazioni dell'intelligenza artificiale
- COBOL: applicazioni gestionali
- C: ideato per la scrittura del sistema operativo Unix, consente di esprimere elaborazioni di "basso livello" (per es., sui bit delle celle di memoria)
- Java: applicazioni Internet
- Matlab: calcolo numerico matriciale

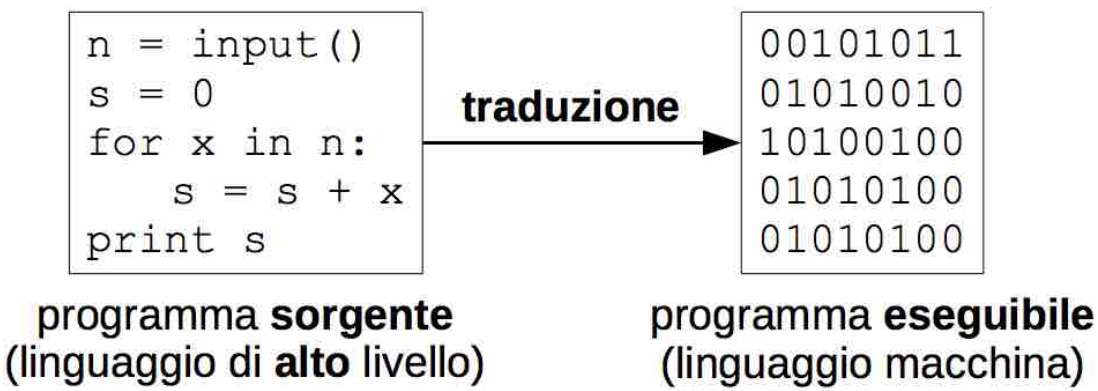
Linguaggi di alto livello

I linguaggi esistenti possono essere suddivisi (in modo non mutuamente esclusivo) nelle seguenti categorie, in base al modo in cui consentono di rappresentare gli algoritmi e i dati da elaborare

- **imperativi/procedurali**: descrivono gli algoritmi come **sequenze** di **istruzioni** (operazioni) da **eseguire** sui dati di ingresso e sui risultati intermedi, per ottenere il risultato desiderato (es.: FORTRAN, C, Python)
- **dichiarativi** (logici, funzionali): descrivono mediante un formalismo logico-matematico il **risultato** che si vuole ottenere, non **come** ottenerlo (es.: Lisp, Python)
- **orientati agli oggetti**: rappresentano i dati come "oggetti" appartenenti a "classi" caratterizzate da certe "proprietà", ovvero valori e operazioni eseguibili sulle loro istanze (es.: SmallTalk, C++, Java, Python)

Traduzione in linguaggio macchina

Un programma scritto in un linguaggio di alto livello è detto programma **sorgente**, e non è "comprensibile" (e quindi non è eseguibile) da un calcolatore.
Per renderlo **eseguibile** è necessario tradurlo in linguaggio macchina.
Il processo di **traduzione** può a sua volta essere descritto attraverso un algoritmo: esso può quindi essere codificato in linguaggio macchina ed eseguito **dallo stesso calcolatore**.



Linguaggi *compilati* e *interpretati*

Dal punto di vista del processo di traduzione, esistono due categorie di linguaggi di alto livello

- linguaggi **compilati**: l'intero programma sorgente viene tradotto in linguaggio macchina **prima** dell'esecuzione (esempi: Fortran, C, C++, ...)
- linguaggi **interpretati**: ogni singola istruzione del programma sorgente viene tradotta in linguaggio macchina ed eseguita, prima dell'elaborazione dell'istruzione successiva (esempi: Lisp, Basic, **Python**, Matlab, ...); questo rende l'esecuzione di programmi scritti in un linguaggio interpretato più lenta rispetto ai linguaggi compilati

Ambienti di programmazione

Per ogni linguaggio di alto livello sono disponibili uno o più **ambienti di programmazione** (o **ambienti integrati di sviluppo** – *integrated development environment*): un insieme di programmi che supportano gli utenti nello **sviluppo** dei propri programmi (scrittura, verifica, esecuzione).

Componenti principali di un ambiente di programmazione

- ***editor***: facilita la scrittura dei programmi, per esempio evidenziando con colori diversi le diverse componenti (istruzioni, espressioni, ecc.)
- **compilatore** o **interprete**: traduce in linguaggio macchina i programmi o le singole istruzioni, e (nel caso dell'interprete) esegue queste ultime
- ***debugger***: facilita l'identificazione di eventuali errori ("*bug*")

Il linguaggio Python

Sommario

- Introduzione al linguaggio Python
- Gli elementi principali del linguaggio
  - variabili, istruzione di assegnamento, espressioni, tipi di dato (numeri e stringhe); le funzioni `input` e `print`
  - espressioni condizionali; l'istruzione condizionale (`if-else`) e iterativa (`while`); l'istruzione `break`
- Funzioni
  - funzioni di *libreria*, l'istruzione `from-import`, le librerie `math` e `random`
  - definizione di nuove funzioni: le istruzioni `def` e `return`
- Tipi di dato strutturati: stringhe, liste, dizionari; l'istruzione iterativa `for`; definizione di strutture dati
- Lettura e scrittura di dati su *file*
- Esempi di algoritmi: ricerca sequenziale e binaria; ordinamento per selezione e per inserimento

Introduzione al linguaggio Python

Cenni storici e motivazioni

- Il linguaggio Python
  - linguaggio di alto livello, interpretato, *open source*
  - ideato nel 1989 dall'informatico olandese Guido Van Rossum (`https://www.python.org/~guido/`)
  - ideato originariamente come linguaggio di *scripting*, poi evolutosi come linguaggio completo
  - punto di forza: facilità nella scrittura dei programmi
  - sono disponibili diversi ambienti di programmazione
- Perché si usa Python in questo corso
  - sintassi minimale, facile da apprendere
  - possiede gli elementi di base comuni a gran parte dei linguaggi di alto livello: fornisce quindi le basi per l'apprendimento, anche autonomo, di altri linguaggi

Documentazione e risorse

- Sito Web ufficiale (versione italiana):
  `http://www.python.it`
- Ambiente di programazione usato in questo corso:
  `IDLE`, `http://www.python.it/download`
- La versione supportata del linguaggio è la **3**
- Testo di riferimento
  - C. Horstmann, R.D. Necaise, *Concetti di informatica e fondamenti di Python*, Maggioli, 2014 (1a ed.) o 2019 (2a ed.), capitoli 1–8, 12 (disponibile presso la Biblioteca della Facoltà di Ingegneria e Architettura)
  - C. Horstmann, R.D. Necaise, *Python – Introduzione alla programmazione*, Maggioli, 2023 (3a ed. dello stesso testo)

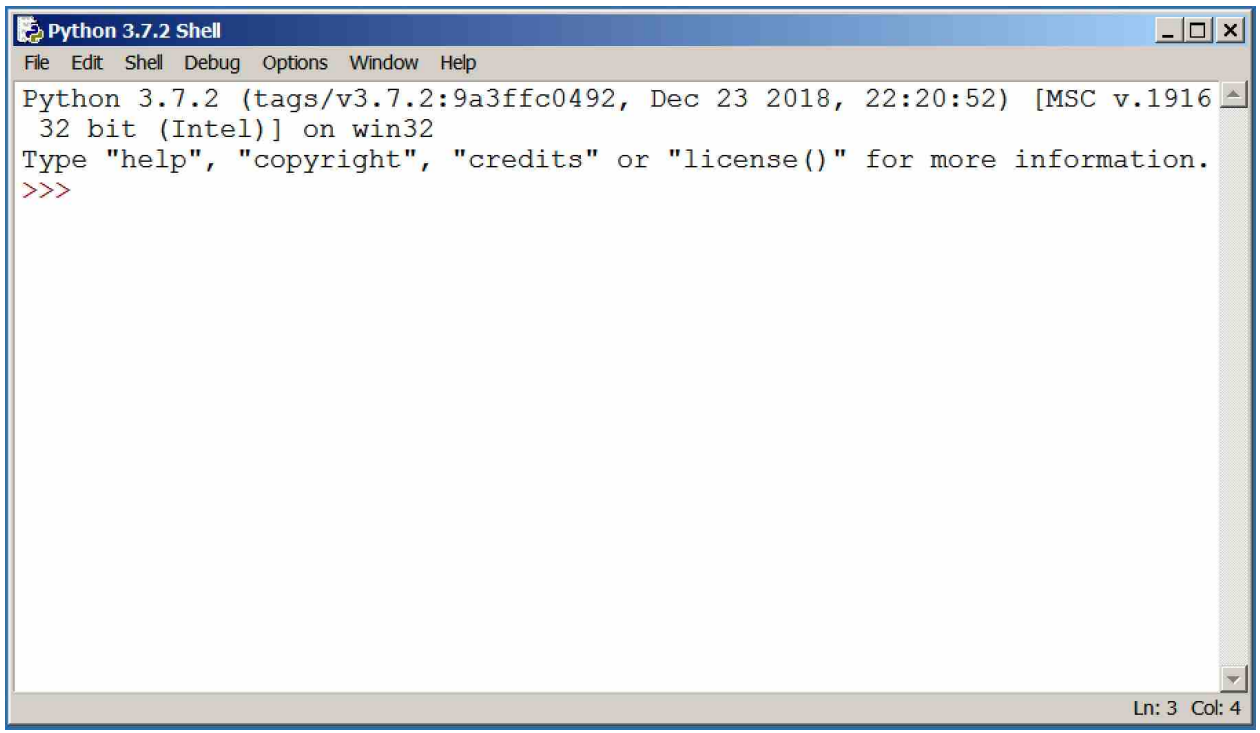
L'ambiente di programmazione IDLE

In questo corso si usa l'ambiente `IDLE`. Componenti principali (comuni a tutti gli ambienti per linguaggi interpretati):

- **interprete**, o ***shell*** (la finestra `Python shell`, che si apre all'avvio di IDLE): consente di eseguire **singole** istruzioni e valutare espressioni in modo interattivo. Il simbolo >>> seguito dal cursore lampeggiante indica che l'interprete è in attesa dell'inserimento di un'istruzione o di un'espressione
- ***editor***: consente di scrivere un programma (**sequenza** d'istruzioni) in una finestra dedicata, memorizzarlo in un *file* di testo, ed eseguirlo in un momento successivo. Per aprire una finestra dell'*editor*, scegliere la voce `New File` dal menu `File`

L'ambiente di programmazione IDLE

Un esempio della finestra della *shell* di `IDLE` che compare all'avvio del programma (il testo mostrato nelle prime righe dipende dalla versione di Python, dallo *hardware* e dal sistema operativo del proprio calcolatore)



Elementi principali del linguaggio Python

Istruzioni, espressioni, variabili

Come per la maggior parte dei linguaggi di alto livello, un programma Python è composto da una sequenza di **istruzioni**.

Tre sono le istruzioni fondamentali

- istruzione di **assegnamento**
- istruzione **condizionale**
- istruzione **iterativa**

Ognuna di esse contiene **espressioni** che producono valori da elaborare (per esempio, espressioni aritmetiche).

In particolare, l'istruzione di assegnamento consente di memorizzare nelle celle di memoria del calcolatore i dati da elaborare e i risultati. Le celle vengono indicate attraverso nomi simbolici, detti **variabili**.

Sono inoltre disponibili **funzioni** per l'**acquisizione** dei dati da elaborare, attraverso dispositivi periferici d'ingresso (*input*) quali la tastiera e la memoria secondaria, e per l'**invio** di dati ai dispositivi d'uscita (*output*), come la memoria secondaria e lo schermo. (Il concetto di *funzione* nei linguaggi di alto livello e in Python verrà spiegato più avanti).

Scrittura ed esecuzione di programmi

Un programma Python è costituito da una **sequenza** d'istruzioni memorizzate in un *file* di testo.

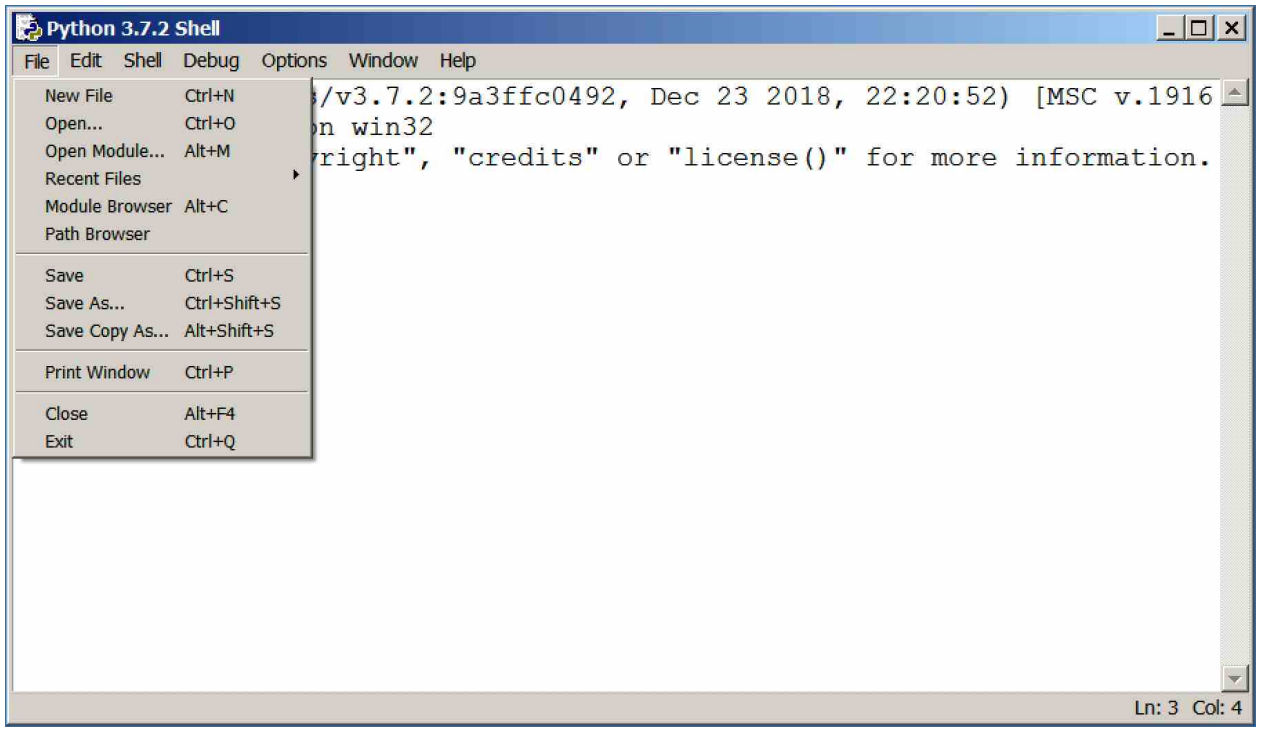
Il programma potrà essere eseguito all'interno di un ambiente di programmazione come `IDLE`, oppure in una *finestra dei comandi* (*shell*) del sistema operativo, purché sia stato installato e configurato un interprete Python.

I nomi dei *file* che contengono programmi Python devono avere estensione `.py` (per esempio, `programma.py`).

Come ogni ambiente di programmazione, `IDLE` comprende un *editor* di testi per facilitare la scrittura di programmi, per esempio evidenziando con colori diversi le diverse componenti sintattiche (nomi delle istruzioni, numeri, stringhe, ecc.).

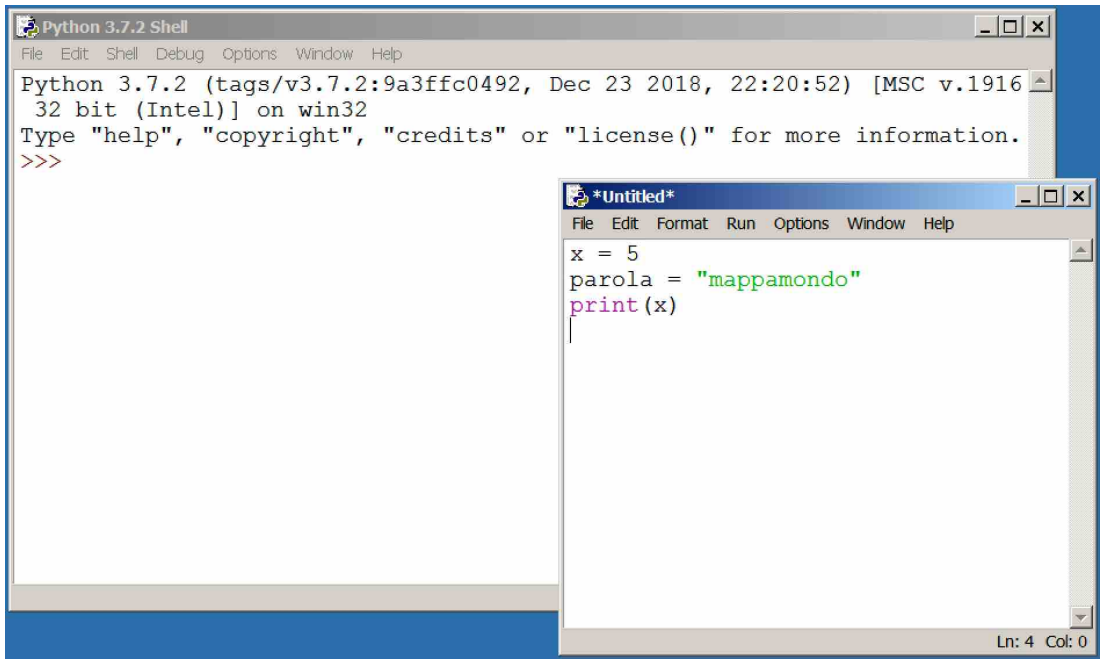
Scrittura ed esecuzione di programmi

Per aprire una nuova finestra dell'*editor* di `IDLE`, scegliere la voce `New File` del menu `File`



Scrittura ed esecuzione di programmi

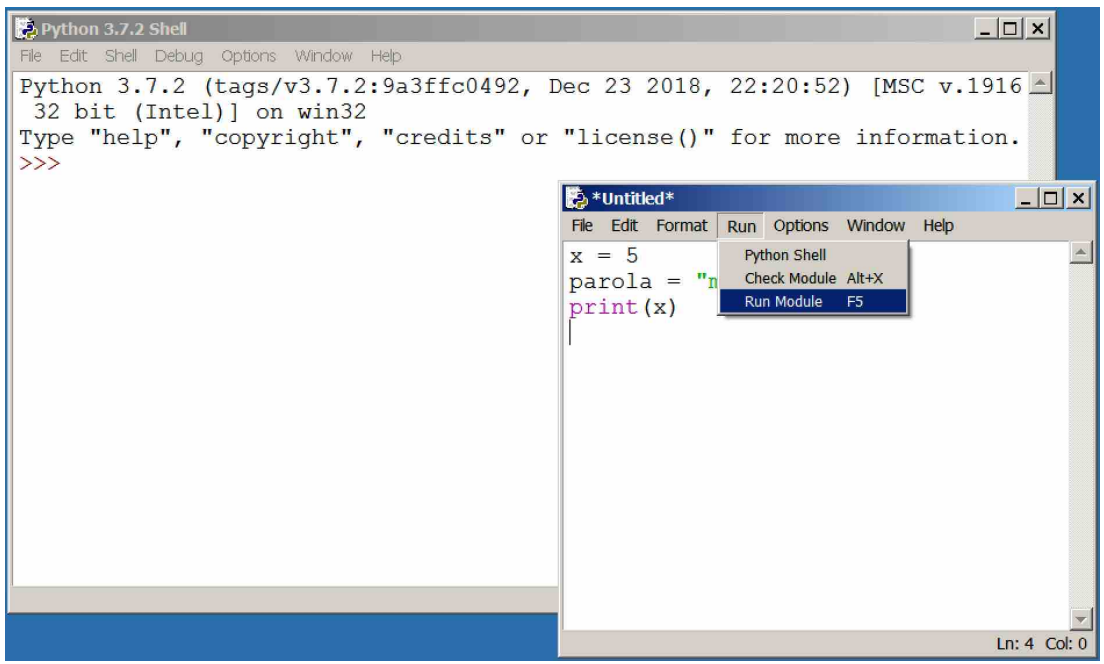
Ogni istruzione di un programma deve essere scritta in una riga **distinta**, e **senza spazi** all'inizio.



Notare i diversi colori usati dall'*editor* dell'ambiente IDLE per evidenziare i diversi elementi delle istruzioni: nomi delle variabili, stringhe, parole riservate del linguaggio (come `print`).

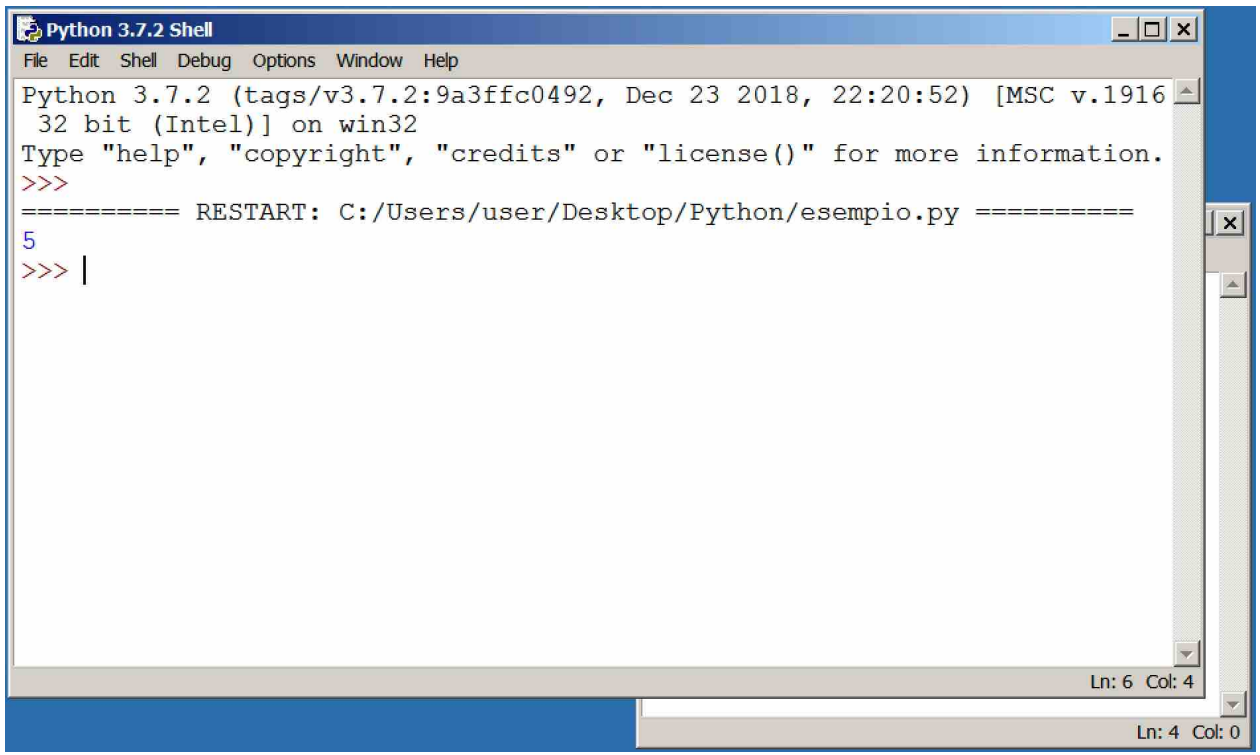
Scrittura ed esecuzione di programmi

Dopo aver memorizzato il programma in un *file*, il cui nome deve avere estensione `.py`, esso potrà essere **eseguito** nell'ambiente IDLE, quando la finestra dell'*editor* che contiene il programma è **attiva**, scegliendo la voce `Run module` del menu `Run` associato alla stessa finestra, oppure premendo il tasto `F5`



Scrittura ed esecuzione di programmi

L'esecuzione di un programma causa il "riavvio" (*restart*) della *shell* (il significato del "riavvio" diventerà chiaro più avanti). I risultati delle elaborazioni verranno poi mostrati sulla stessa *shell*.
Per esempio, in basso si mostra il risultato dell'esecuzione del programma precedente (si noti il messaggio che indica il "riavvio" della *shell*)



Scrittura ed esecuzione di programmi

Qualche precisazione sull'apertura di *file* contenenti programmi Python.

Su alcuni sistemi operativi un doppio "*click*" sull'icona di un tale *file* fa sì che il programma venga aperto in una nuova finestra dell'*editor* di `IDLE` (consentendone la modifica o l'esecuzione, come già visto).

In altri sistemi operativi questa azione comporta invece l'**esecuzione** del programma nella *shell* (finestra dei comandi) dello stesso sistema operativo, che verrà **chiusa** automaticamente non appena il programma sarà terminato.

Se si desidera visualizzare o eseguire un programma all'interno dell'ambiente `IDLE`

- aprire il *file* che lo contiene usando la voce `Open` dal menu `File` di `IDLE`, **oppure**
- selezionare l'icona del *file* con il tasto destro del *mouse* e scegliere tra le opzioni l'apertura dello stesso *file* con il programma `IDLE`

Istruzioni, espressioni, variabili

Nel seguito si descrivono la **sintassi** e la **semantica** delle principali istruzioni ed espressioni del linguaggio Python.

- **Sintassi**: insieme di regole che indicano come istruzioni ed espressioni devono essere **scritte**, ovvero quali sono le istruzioni e le espressioni **corrette** o **valide** del linguaggio
- **Semantica**: insieme di regole che definiscono il **significato** delle istruzioni e delle espressioni valide, ovvero come esse devono essere **eseguite** (istruzioni) o **valutate** (espressioni) dal calcolatore

La sintassi verrà descritta usando

- il colore nero,
  `e questo tipo di carattere`,
  per le parti da scrivere in modo testuale
- il colore rosso per indicare le parti da sostituire come spiegato caso per caso

Un suggerimento per lo studio

La comprensione della sintassi e della semantica di un linguaggio di programmazione è un prerequisito **indispensabile** per la codifica di algoritmi (ovvero la scrittura di programmi) in tale linguaggio.

In altre parole, prima di cimentarsi nella scrittura di programmi è necessario essere in grado di comprendere (ovvero, saper eseguire, per esempio con l'ausilio di carta e matita) un qualsiasi programma, come quelli mostrati di seguito.

Variabili e istruzione di assegnamento

Nei programmi in linguaggio macchina i dati d'ingresso e i risultati delle elaborazioni devono essere memorizzati in **celle di memoria** scelte dal programmatore, alle quali ci si riferisce tramite il loro **indirizzo** numerico.

Anche i linguaggi di alto livello prevedono la memorizzazione di dati e risultati nelle celle di memoria. Questo avviene attraverso l'istruzione di **assegnamento**, una delle istruzioni fondamentali e comuni alla maggior parte di tali linguaggi.

Contrariamente al linguaggio macchina, le celle di memoria vengono indicate nei linguaggi di alto livello attraverso **nomi simbolici** scelti dal programmatore.

Tali nomi simbolici sono detti **variabili**, poiché i valori memorizzati nelle celle corrispondenti possono essere modificati da successive istruzioni di assegnamento.

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 (il significato diventerà chiaro più avanti):

```
x = -3.2
messaggio = "Buongiorno"
y = x + 1
```

Variabili e istruzione di assegnamento: sintassi

Limitazioni sui nomi delle variabili

- possono essere composti da uno o più dei seguenti caratteri
  - lettere minuscole e maiuscole (anche accentate)
  - cifre
  - il carattere _ (*underscore*)

  esempi: `x`, `somma`, `età`, `Massimo_Comun_Divisore`, `y_1`
- **non** devono iniziare con una cifra;
  esempio: `12abc` non è un nome ammesso
- **non** devono coincidere con i nomi predefiniti delle istruzioni e di altri elementi del linguaggio, come per esempio:
  `if`, `while`, `print`

Variabili e istruzione di assegnamento: semantica

L'istruzione di assegnamento viene **eseguita** in due fasi

1. l'interprete valuta espressione, cioè ne determina il valore
2. il valore di espressione viene assegnato (o associato) a variabile

Dopo l'assegnamento, il nome della variabile potrà essere usato in espressioni all'interno di altre istruzioni: nella valutazione di tali espressioni l'interprete sostituirà al nome della variabile il valore a essa associato (si veda più avanti).

L'effetto concreto di un'istruzione di assegnamento è la memorizzazione del valore di espressione all'interno di una o più celle di memoria.

Se un'istruzione di assegnamento si riferisce a una varilabile alla quale in precedenza era già stato assegnato un valore, il nuovo valore **sostituirà** quello precedente.

Espressioni e tipi di dato

Le espressioni Python possono elaborare e produrre valori appartenenti a tre categorie principali, dette **tipi di dato** (più avanti verranno presentati ulteriori tipi di dato Python)

- numeri interi
- numeri frazionari
- sequenze di caratteri

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

- espressioni aritmetiche
- **stringhe** (sequenze di caratteri)

Espressioni aritmetiche

La più semplice espressione aritmetica è un singolo numero.
I numeri vengono rappresentati nelle istruzioni Python in base dieci, con diverse possibili notazioni.

Come altri linguaggi, Python distingue tra due tipi di dato 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` (2×10^3)

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

Espressioni aritmetiche

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:

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

Espressioni aritmetiche: esempi

Alcuni esempi di istruzioni di assegnamento contenenti espressioni aritmetiche. Notare che ciò che viene assegnato a ciascuna variabile è il **valore** dell'espressione corrispondente, indicato in neretto sulla destra.

```
x = -5                          -5
y = 1 + 1                        2
z = (1 + 2)*3                    9
circonferenza = 2*3.14*3        18.84
q1 = 6/2                        3.0
q2 = 7.5/3                      2.5
q3 = 5//2                        2
resto = 10 % 2                   0
```

Espressioni aritmetiche: osservazioni

- 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`

- Il risultato prodotto dall'operatore / è sempre rappresentato come valore frazionario. Esempi:

  `5/2` $\rightarrow$ `2.5`
  `4/2` $\rightarrow$ `2.0`
  `3/3` $\rightarrow$ `1.0`

Espressioni aritmetiche: osservazioni

L'operatore `//` produce il più grande **intero** non maggiore del quoziente della divisione tra i suoi operandi. Se entrambi gli operandi sono interi tale valore viene rappresentato come intero, 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`

Espressioni che elaborano stringhe

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

```
"Questa è una stringa"
'Questa è una stringa'
"A"
"qwerty, 123456"
""  (una stringa vuota)
```

È quindi possibile assegnare una stringa a una variabile, come negli esempi che seguono:

```
testo = "Questa è una stringa"
carattere = "a"
messaggio = "Premere un tasto per continuare."
t = ""
```

Espressioni che elaborano 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 stringhe qualsiasi.

Esempi

- `parola = "mappa" + "mondo"`
  assegna alla variabile `parola` la stringa `"mappamondo"`
- `testo = "Due" + " " + "parole"`
  assegna alla variabile `testo` la stringa `"Due parole"`
  (notare che la stringa `" "` è composta da un carattere di spaziatura)

Espressioni contenenti nomi di variabili

Come caso particolare, anche il nome di una variabile alla quale sia già stato assegnato un valore costituisce un'espressione.

Il valore di una tale espressione coincide con il valore che è associato alla stessa variabile nel momento in cui l'espressione viene valutata dall'interprete (ovvero, coincide con l'ultimo valore assegnato a tale variabile in ordine di tempo, nel caso di più assegnamenti).

Per esempio, dopo l'esecuzione della sequenza d'istruzioni

```
x = 5
y = x
```

alla variabile `y` sarà assegnato il valore `5`.

Espressioni contenenti nomi di variabili

Da quanto detto sopra segue che se a una variabile è già stato assegnato un valore, il suo nome potrà essere usato come **operando** all'interno di un'espressione. Quando l'interprete valuterà tale espressione, sostituirà al nome della variabile il valore a essa associato. Esempi

- eseguendo le istruzioni

```
raggio = 2
circonferenza = raggio*6.28
```

  alla variabile `circonferenza` sarà assegnato il valore `12.56`

- eseguendo le istruzioni

```
parola1 = "mappa"
parola2 = parola1 + "mondo"
```

  alla variabile `parola2` sarà assegnato il valore `"mappamondo"`

Espressioni contenenti nomi di variabili

Si consideri questa sequenza di istruzioni:

```
x = 5
y = x
x = 1
```

Da quanto si è detto in precedenza segue che il valore associato alla variabile `x` dopo l'esecuzione di tali istruzioni sarà `1`.

Ci si può però chiedere quale sarà il valore associato alla variabile `y`: `5` oppure `1`?

In casi come questo nei programmi Python il valore associato a `y` resta immutato (in questo esempio resterà pari a `5`), in quanto l'ultima istruzione ha come unico effetto la modifica del valore associato a `x`.

Espressioni contenenti nomi di variabili

Un'espressione contenente il nome di una variabile alla quale **non** sia stato ancora assegnato nessun valore non ha significato, ed è considerata errata dall'interprete.

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

```
x = h + 1
```

causerà la terminazione del programma da parte dell'interprete e la stampa di un messaggio d'errore nella *shell*, come si vedrà più avanti.

Programmi e istruzioni di assegnamento

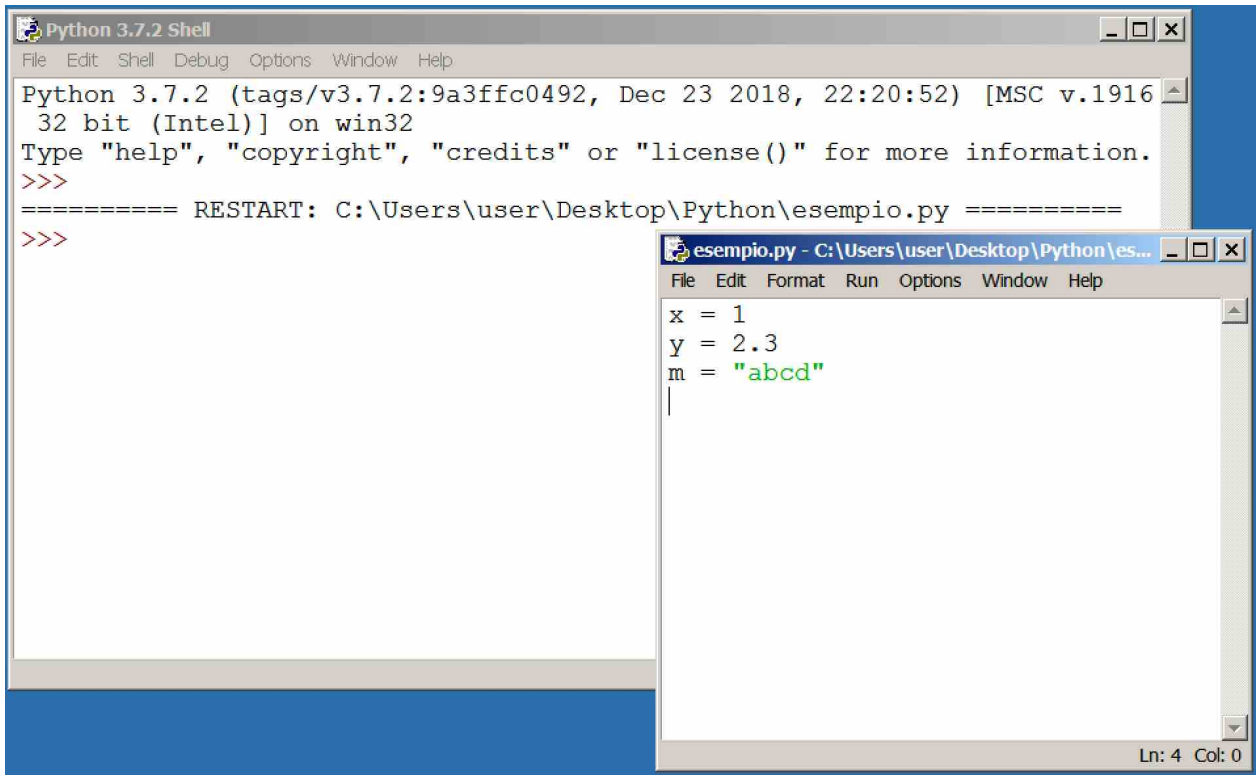
Si noti che le istruzioni di assegnamento non producono nessun effetto **visibile** nella *shell* durante l'esecuzione di un programma (a meno che tali istruzioni contengano errori).

Si consideri come esempio il programma seguente

```
x = 1
y = 2.3
m = "abcd"
```

Il contenuto della *shell* dopo la sua esecuzione è mostrato di seguito.

Programmi e istruzioni di assegnamento



La funzione `print`

Ogni linguaggio di programmazione prevede specifiche istruzioni o **funzioni** per l'acquisizione dalle periferiche d'ingresso dei valori da elaborare e l'invio dei risultati alle periferiche d'uscita. (Il concetto di *funzione* nei linguaggi di alto livello verrà descritto più avanti).

Nel linguaggio Python il meccanismo più semplice per comunicare agli utenti di un programma i risultati delle elaborazioni consiste nello stampare nella *shell* dell'ambiente di programmazione (o del sistema operativo) il **valore** di una o più **espressioni**, attraverso la funzione `print`.

La funzione `print`: sintassi e semantica

Sintassi: `print(`espressione`)`

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

Semantica: viene stampato nella *shell* il **valore** di espressione. Poiché anche i nomi delle variabili costituiscono espressioni, la funzione `print` consente di stampare il valore associato a qualsiasi variabile.

È anche possibile stampare i valori di un numero qualsiasi di espressioni, con la seguente sintassi:

`print(`espressione$_1$`, `espressione$_2$`, ...)`

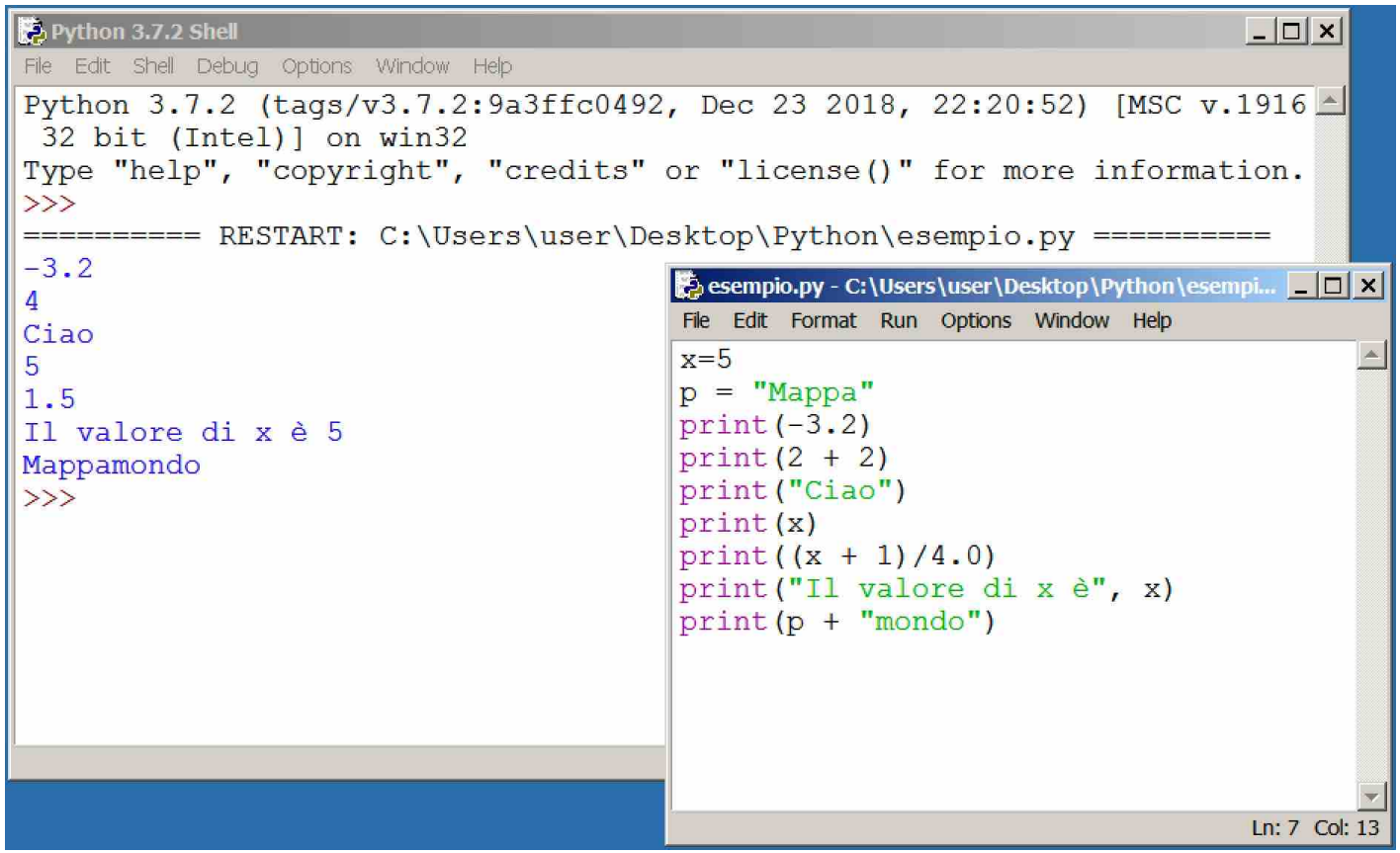
In questo caso i valori delle espressioni vengono stampati su una stessa riga, separati da un carattere di spaziatura.

La funzione `print`: esempi

Il programma che segue ha come effetto **visibile** la stampa nella *shell* di una sequenza di valori. L'esito della sua esecuzione è mostrato nella pagina seguente.

```
x = 5
p = "Mappa"
print(-3.2)
print(2 + 2)
print("Ciao")
print(x)
print((x + 1)/4.0)
print("Il valore associato a x è", x)
print(p + "mondo")
```

La funzione `print`: esempi



Caratteri speciali nelle stringhe

- Per inserire in una stringa un apice singolo o doppio è necessario racchiuderla tra apici dell'altro tipo, oppure far precedere l'apice dal carattere \ (detto *backslash*), senza spazi tra i due. Per esempio (il risultato è mostrato nella pagina seguente):

```
print("L'apostrofo")
print('L\'apostrofo')
print('Doppi "apici" in una stringa')
print("Doppi \"apici\" in una stringa")
```

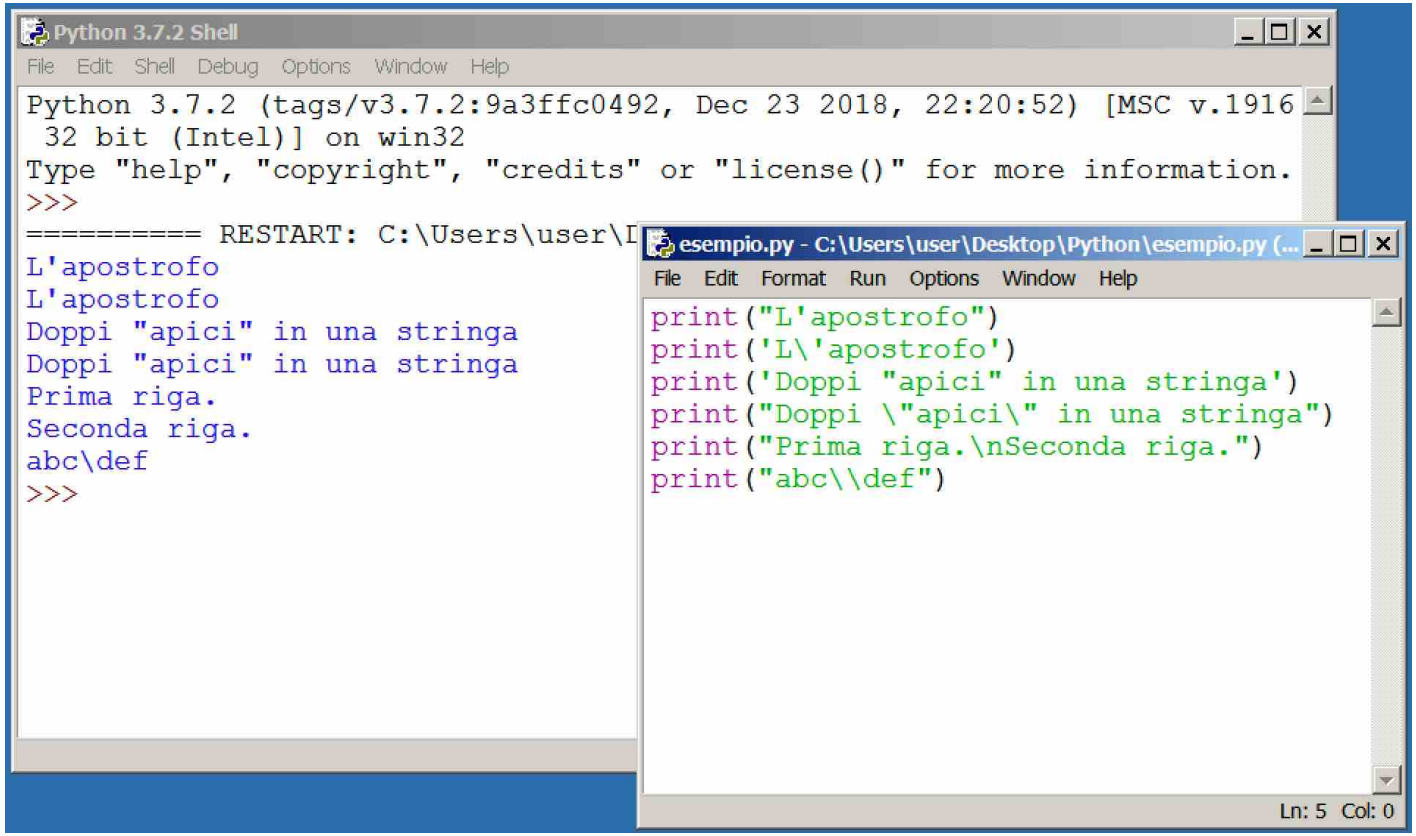
- Per rappresentare un'interruzione di riga ("a capo") all'interno di una stringa, si usa la sequenza di caratteri `\n`, detta *newline*. Il carattere *newline* produce un'interruzione di riga quando la stringa viene stampata. Esempio:

```
print("Prima riga.\nSeconda riga.")
```

- Dato il significato particolare del carattere \ nelle stringhe, per stamparlo testualmente si deve scrivere `\\`. Esempio:

```
print("abc\\def")
```

Caratteri speciali nelle stringhe: esempi



Esercizi

Che cosa viene stampato nella *shell* dall'esecuzione del seguente programma?

```
print(2 - 1)
print("2 - 1")
print("2\n1")
```

Che cosa viene stampato nella *shell* dall'esecuzione del seguente programma?

```
b = 2
B = 3
h = 2.5
print("Area del trapezio: ", (b + B)*h/2)
```

La funzione `input`

In molti casi i valori dei dati da elaborare non sono noti nel momento della **scrittura** di un programma, ma devono essere acquisiti **durante la sua esecuzione**, attraverso periferiche d'ingresso come la tastiera o la memoria secondaria.

Nei programmi Python l'acquisizione di dati mediante la tastiera si ottiene per mezzo della funzione `input`.

Questa funzione consente all'utente, durante l'esecuzione di un programma, di immettere una qualsiasi sequenza di caratteri nella *shell*; alla pressione del tasto `INVIO` la sequenza inserita sarà restituita dall'interprete sotto forma di **stringa**.

Poiché di norma i dati acquisiti durante l'esecuzione di un programma devono essere memorizzati in variabili per essere successivamente elaborati, la funzione `input` compare spesso nelle istruzioni di assegnamento.

La funzione `input`: sintassi e semantica

Di seguito si descrive l'uso di `input` come parte di un'istruzione di assegnamento.

Sintassi: variabile = `input`(espressione)
Si noti che espressione è opzionale (può cioè non essere presente).

Semantica: l'interprete stampa nella *shell* espressione (se presente), che di norma è una **stringa**, e resta in attesa che l'utente inserisca nella *shell*, attraverso la tastiera, una sequenza di caratteri fino alla pressione del tasto `INVIO`; tale sequenza viene assegnata a variabile sotto forma di **stringa**.

Notare che durante l'esecuzione della funzione `input` l'esecuzione del programma resta **sospesa** fino alla pressione del tasto `INVIO`.

La funzione `input` come espressione

Sintatticamente il costrutto `input(`espressione`)` costituisce un tipo particolare di **espressione** Python (detta **chiamata di funzione**, come si vedrà più avanti). Infatti, analogamente alle altre espressioni, esso ha lo scopo di produrre un valore che dovrà poi essere elaborato dal programma (nel caso qui considerato, tale valore viene assegnato a una variabile).

La stringa messaggio viene di norma usata per comunicare all'utente che il programma è in attesa di ricevere un certo dato d'ingresso.

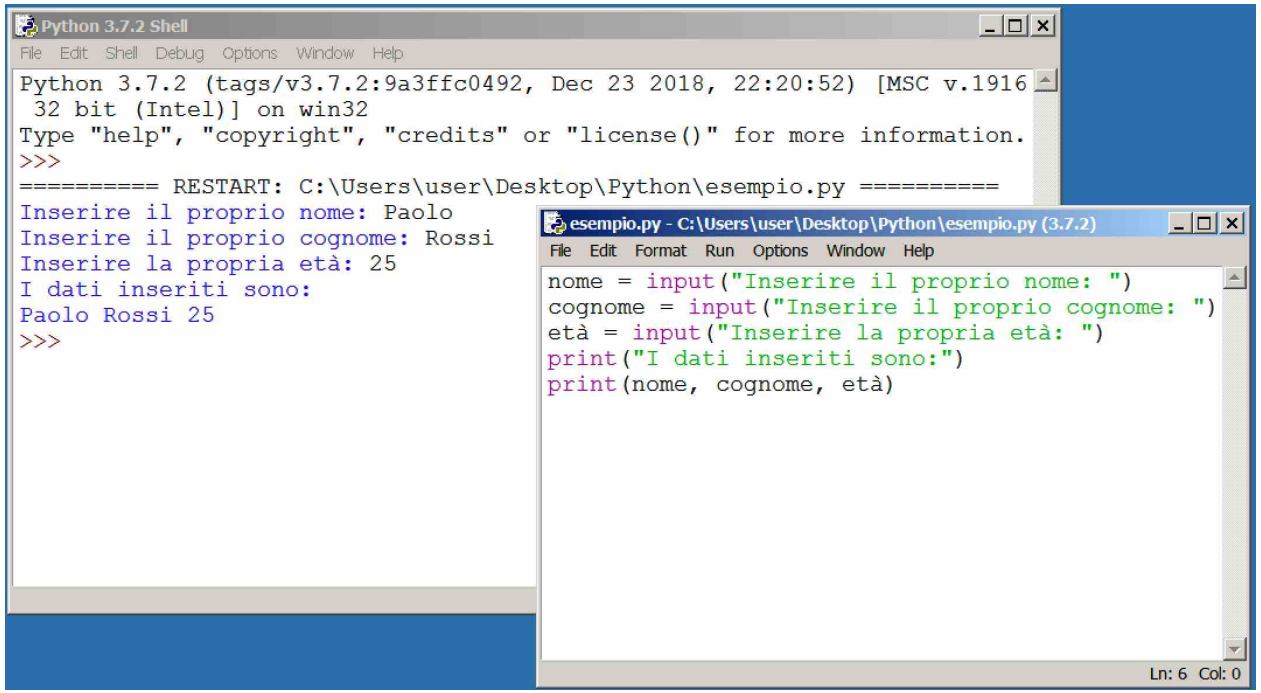
La funzione `input`: esempio

Il programma che segue acquisisce tre sequenze di caratteri e le stampa nella *shell*.

```
nome = input("Inserire il proprio nome: ")
cognome = input("Inserire il proprio cognome: ")
età = input("Inserire la propria età:  ")
print("I dati inseriti sono:")
print(nome, cognome, età)
```

L'esito della sua esecuzione è mostrato nella pagina seguente. Si noti che le stringhe facenti parte delle chiamate della funzione `input` vengono stampate nella *shell* in colore blu, mentre le sequenze di caratteri mostrate in nero sono state immesse nella stessa *shell* **durante l'esecuzione** del programma.

La funzione `input`: esempio



Acquisizione di dati tramite `input`: limitazioni

La funzione `input` consente di acquisire valori di tipo **stringa** (sequenze di caratteri). In un programma può però essere necessario acquisire ed elaborare dati di tipo diverso, per esempio numeri.

Si consideri ancora il programma dell'esempio precedente e si assuma di voler banalmente stampare anche l'età che la persona che immette i dati avrà *dopo un anno*, nel modo seguente

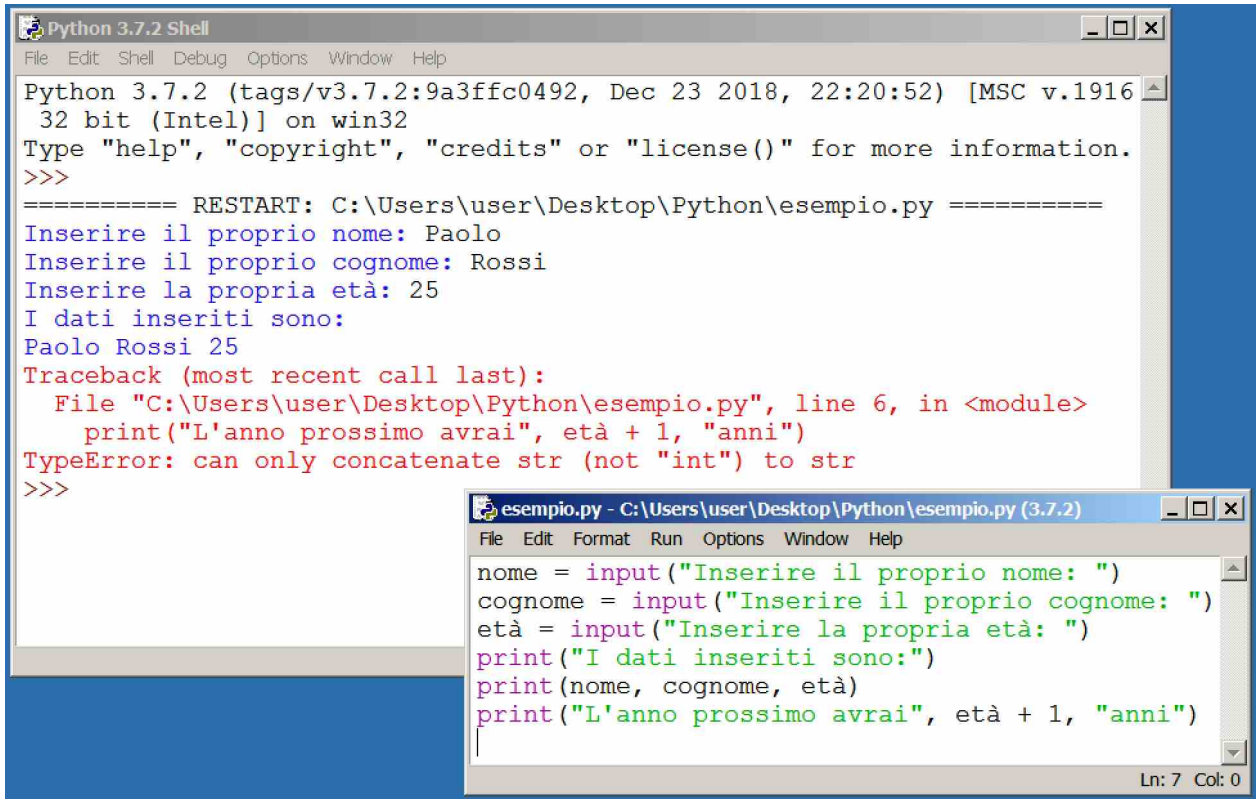
```
print("L'anno prossimo avrai", età + 1, "anni")
```

Come si vede dalla pagina seguente, la valutazione dell'espressione `età + 1` produce un errore: alla variabile `età` è infatti associata una **stringa**, non un numero.

Lo stesso errore si otterrebbe, per esempio, scrivendo:

```
x = "1" + 1
```

Acquisizione di dati tramite `input`: limitazioni



La funzione `eval`

Se una stringa contiene una qualsiasi espressione **valida** del linguaggio Python, la funzione `eval` consente di ottenere il valore di tale espressione.

Assumendo che x sia un'espressione di tipo **stringa**, la sintassi della chiamata di `eval` è la seguente:

```
eval(x)
```

Per esempio, l'espressione `eval("1+1")` produce il valore 2.

Se la stringa x non rappresenta un'espressione valida (per esempio, `eval("1+*d-")`), la valutazione di `eval(x)` produce un errore.

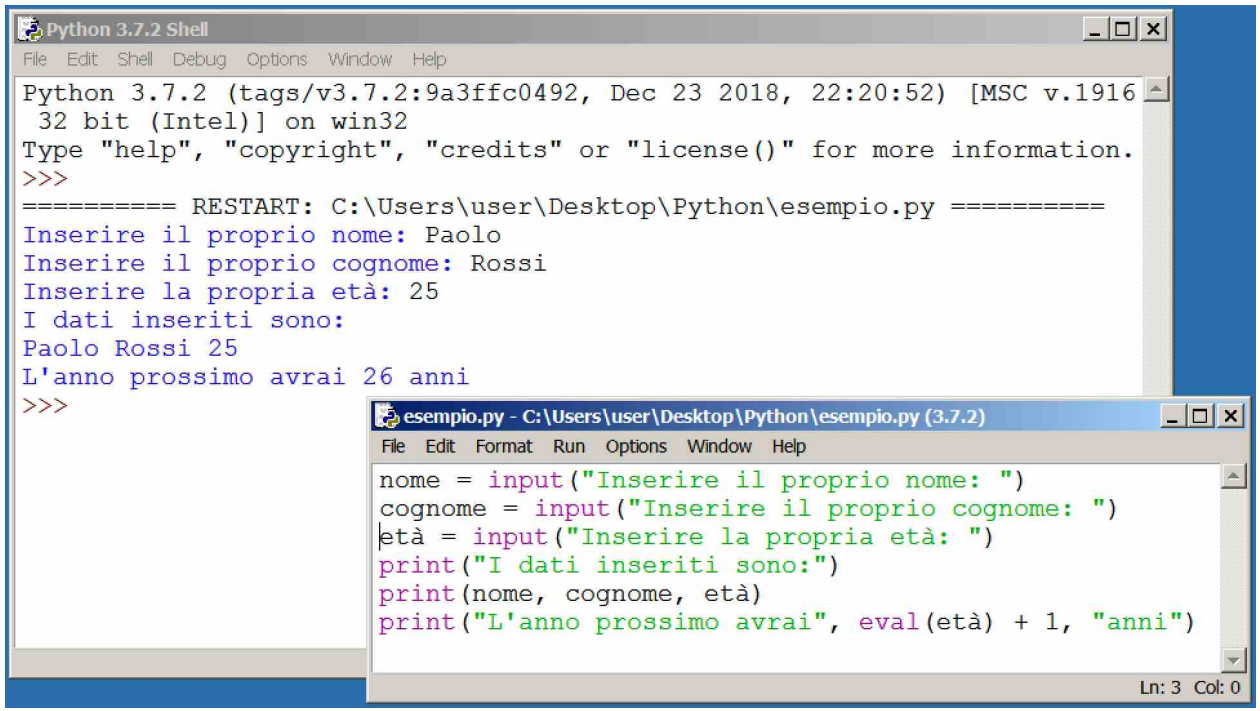
La funzione `eval` può quindi essere usata per "convertire" in valori numerici i dati acquisiti mediante la funzione `input`.

La funzione `eval`: esempio

Versione corretta del programma precedente: si noti l'uso di `eval` per convertire l'età in un valore numerico

```
nome = input("Inserire il proprio nome: ")
cognome = input("Inserire il proprio cognome: ")
età = input("Inserire la propria età:  ")
print("I dati inseriti sono:")
print(nome, cognome, età)
print("L'anno prossimo avrai", eval(età) + 1, "anni")
```

La funzione `eval`: esempio



La funzione `eval`

La funzione `eval` può anche essere usata in combinazione con `input`, in un'unica espressione.

Per esempio, nel programma precedente si sarebbe potuto scrivere

```
età = eval(input("Inserire la propria età:  "))
```

e poi

```
print("L'anno prossimo avrai", età + 1, "anni")
```

Errori di sintassi

I linguaggi di programmazione sono linguaggi **formali**, la cui sintassi è definita da regole precise e inderogabili.

I calcolatori non sono dotati di "buon senso": qualsiasi istruzione o espressione che non rispetti le regole di sintassi di un linguaggio, anche per un minimo dettaglio, sarà "incomprensibile" per l'inteprete o il compilatore dello stesso linguaggio. In un linguaggio interpretato come Python ciò causerà l'interruzione del programma e la visualizzazione di un messaggio d'errore.

Nell'esecuzione dei programmi Python gli eventuali messaggi d'errore vengono stampati nella *shell* dell'ambiente di programmazione o del sistema operativo, oppure vengono mostrati nella finestra dell'*editor* dell'ambiente di programmazione.

Di seguito si mostrano alcuni esempi di comuni errori di sintassi nei programmi Python.

Errori di sintassi

Un'espressione contenente il nome di una variabile alla quale **non** sia stato ancora assegnato nessun valore è considerata errata dall'interprete.

Per esempio, l'esecuzione di un programma composto solo dalla seguente istruzione di assegnamento

```
x = h + 1
```

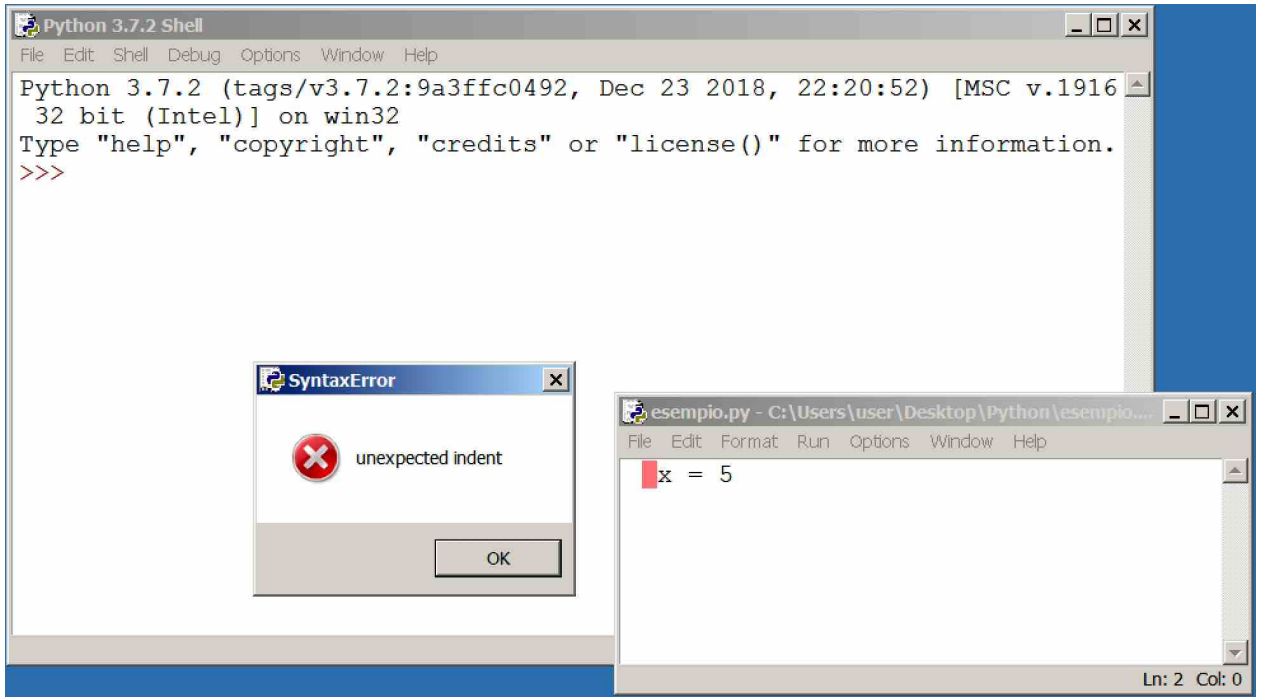
causerà il messaggio d'errore

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

Errori di sintassi

Non è consentito inserire spazi **all'inizio** di una qualsiasi riga di un programma (con le eccezioni descritte più avanti).

Se ciò avviene, quando il programma verrà eseguito all'interno dell'ambiente `IDLE`, comparirà un messaggio d'errore nella finestra dell'*editor*, come mostrato in questo esempio:



Errori di sintassi

Altri esempi di istruzioni o espressioni contenenti errori di sintassi, facilmente comprensibili (si provi a scriverle in un programma e a eseguire quest'ultimo)

- `x =`
- `n = 2 +`
- `a 1 + 1`
- `prin("Buongiorno")`
- `print("Buongiorno"`
- `messaggio = "Buongiorno`

Uso interattivo della *shell*

Come per tutti i linguaggi interpretati, la *shell* non è solo uno strumento per visualizzare i risultati di un programma e immettere i dati d'ingresso, ma è anche uno strumento **interattivo** che consente all'utente di

- scrivere una **singola** istruzione, che verrà eseguita dall'interprete alla pressione del tasto `INVIO`
- scrivere una **singola** espressione, che verrà valutata dall'interprete alla pressione del tasto `INVIO`

Così come nell'*editor*, istruzioni ed espressioni scritte nella *shell* **non** devono essere precedute da **spazi**.

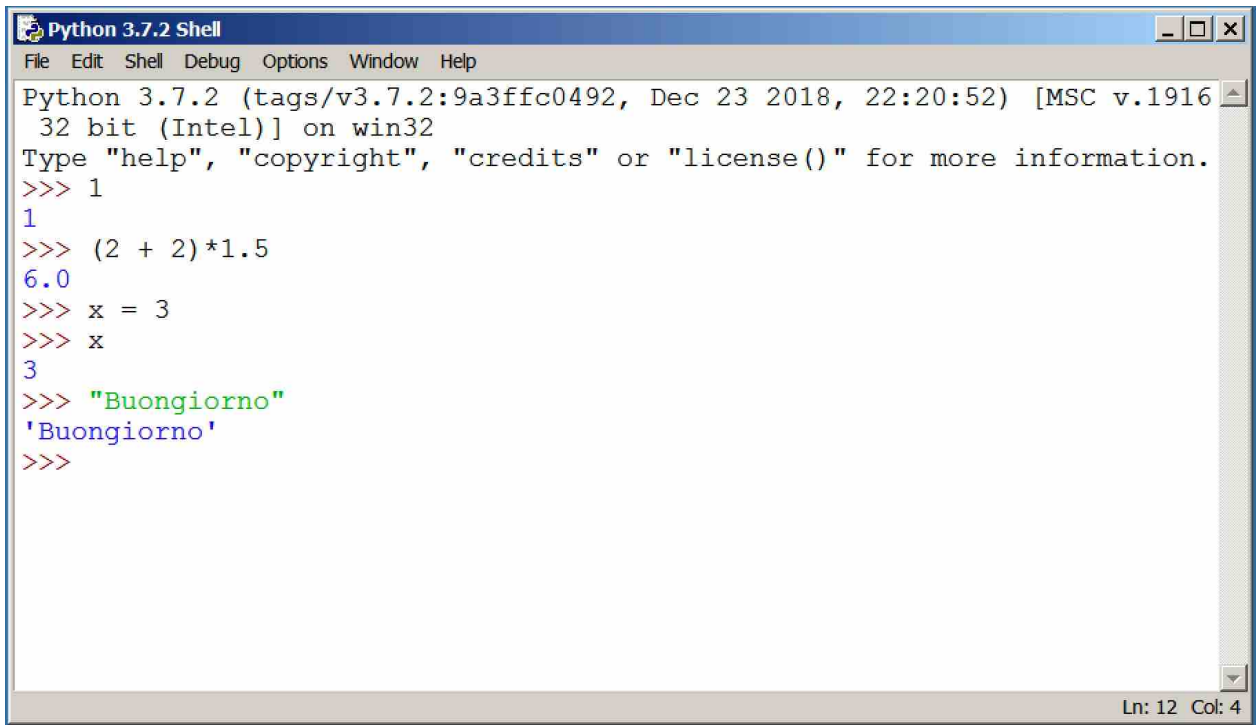
Uso interattivo della *shell*

L'uso interattivo della *shell* è utile per diversi scopi, per esempio

- verificare l'esito dell'esecuzione di una singola istruzione, o il valore di una singola espressione
- visualizzare il valore associato a una variabile (si ricordi che i nomi delle variabili sono espressioni)
- usare la *shell* di Python come una calcolatrice con memoria

Uso interattivo della *shell*: esempi

La figura che segue mostra un esempio di sessione interattiva con la *shell*. Le righe che iniziano con il *prompt* (>>>) contengono espressioni e istruzioni scritte dall'utente; le altre (di colore blu) contengono i valori delle espressioni e quelli stampati mediante la funzione `print`



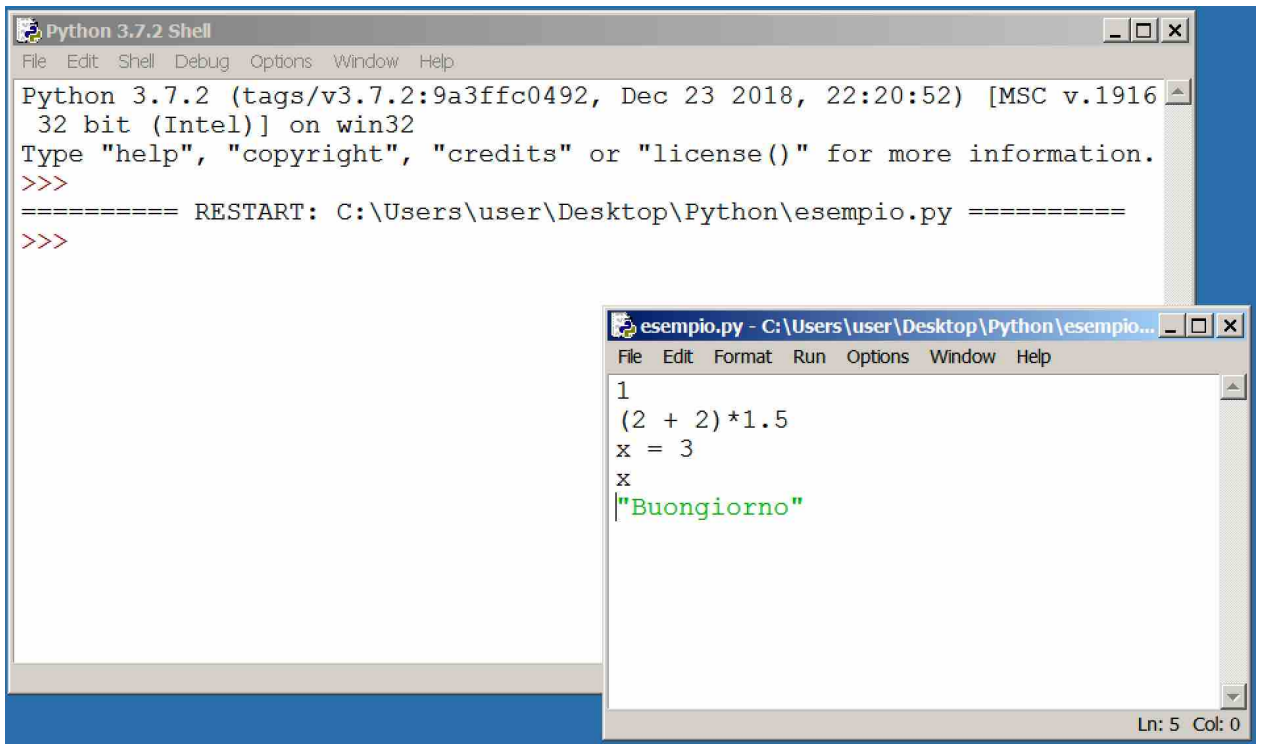
Uso delle funzioni `print` e `input`

Si osservi che nella *shell* **non è necessario** usare la funzione `print` per stampare il valore di un'espressione, né usare `input` per acquisire dati.

Queste due funzioni sono invece **necessarie** nei programmi.

In particolare, in un programma è sintatticamente lecito scrivere in una riga una singola espressione (invece che un'istruzione), ma la sua valutazione non produrrà nessun effetto; in particolare, il suo valore non verrà stampato nella *shell*, come mostrato nell'esempio che segue. La funzione `print` è quindi l'unico strumento utilizzabile **nei programmi** per stampare dati o messaggi nella *shell*.

Uso delle funzioni `print` e `input`: esempio



Osservazioni sull'esecuzione dei programmi

Dopo l'esecuzione di un programma le variabili definite al suo interno non vengono "cancellate", ma restano accessibili dalla *shell*.

Per esempio, si consideri il programma che segue, nel quale viene definita una variabile di nome `x`

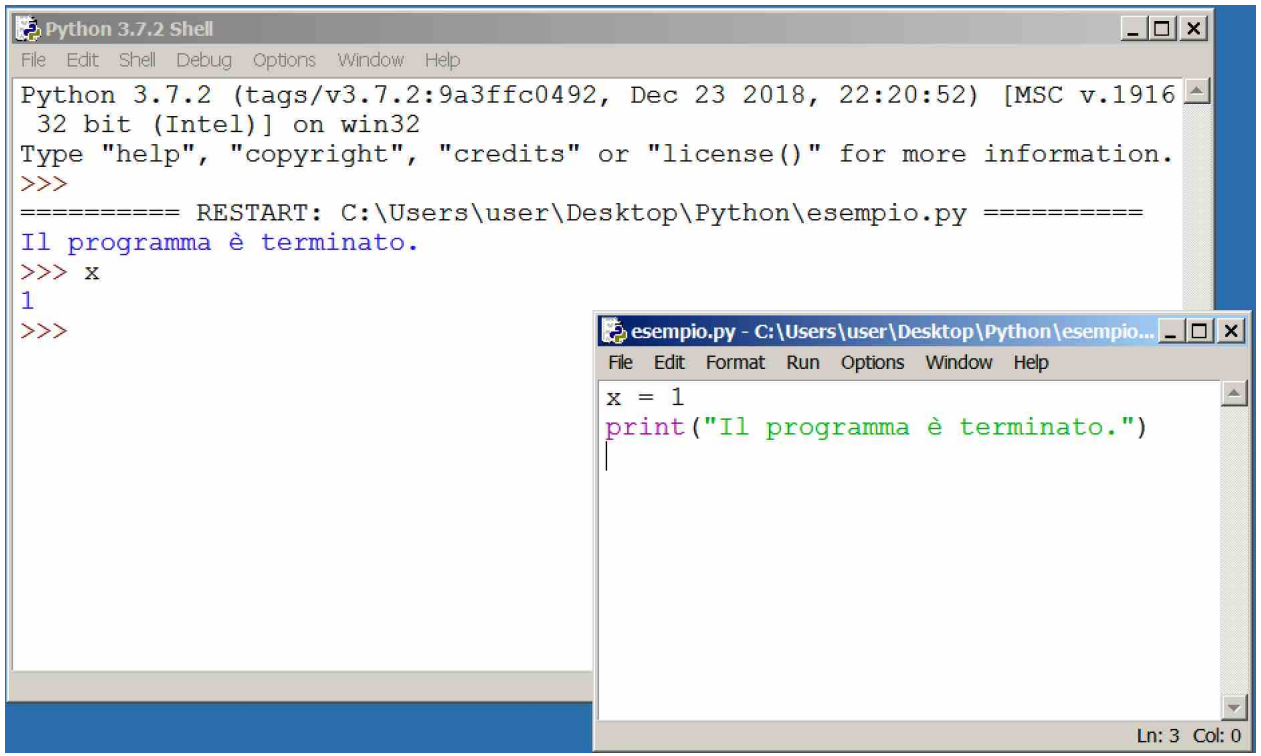
```
x = 1
print("Il programma è terminato.")
```

Dopo l'esecuzione, si valuti nella *shell* l'espressione

```
x
```

Il risultato sarà il valore memorizzato nella variabile `x` dallo stesso programma.

Osservazioni sull'esecuzione dei programmi: esempio



Osservazioni sull'esecuzione dei programmi

Si è detto in precedenza che l'esecuzione di un programma comporta il "riavvio" della *shell*.

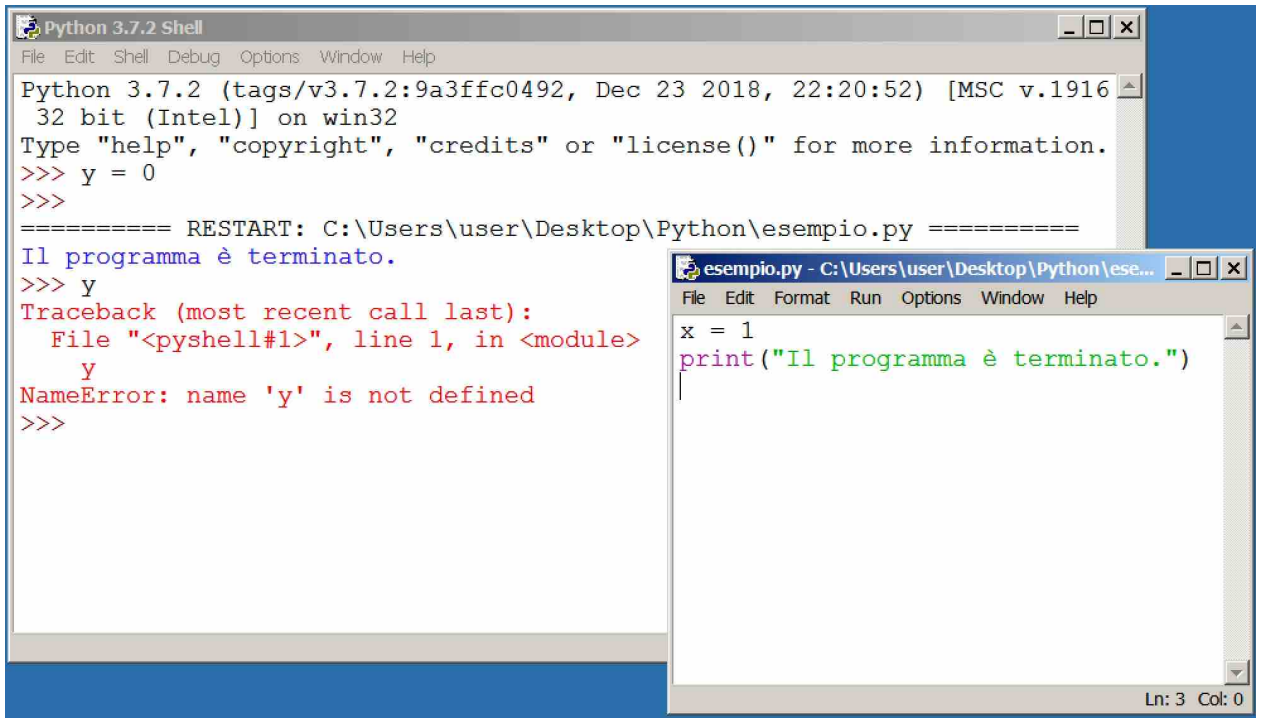
Una delle conseguenze è la "cancellazione" delle eventuali variabili definite precedentemente dall'esecuzione di un altro programma, o da istruzioni scritte nella stessa *shell*.

Per esempio, si provi a scrivere nella *shell* l'assegnamento

```
y = 0
```

Si esegua poi il programma dell'esempio precedente, e si valuti infine nella *shell* l'espressione `y`: come si vedrà, la variabile `y` risulterà non definita.

Osservazioni sull'esecuzione dei programmi: esempio



Esercizi di riepilogo

Determinare il valore delle seguenti **espressioni** Python (verificare successivamente la propria risposta mediante la *shell*)

- `1.0 + 2`
- `(1 + 2)/2`
- `2**3`
- `3//(4*3)`
- `256 % 2`
- `2 % 256`
- `1/0`
- `"2 + 2"`
- `"2" + "2"`

Esercizi di riepilogo

Quali sono i valori delle variabili `x` e `y` dopo la seguente sequenza di istruzioni? (verificare la propria risposta scrivendo le istruzioni nella *shell*, e valutando poi le espressioni `x` e `y`)

```
y = 2
x = y + 1
y = 0
```

Qual è il valore associato alla variabile `c` dopo le seguenti istruzioni?

```
a = "1"
b = "2"
c = a + b
```

Qual è il valore associato alla variabile `k` dopo le seguenti istruzioni?

```
k = 0
k = k + 1
```

Esercizi di riepilogo

Qual è il valore associato alla variabile `p` dopo l'esecuzione delle seguenti istruzioni, se il valore inserito attraverso la tastiera nel momento in cui viene valutata l'espressione `input` è 4?

```
m = 2.5
n = eval(input("Inserire un numero: "))
p = m*n + 1
```

Quali sono i valori delle variabili `r`, `s` e `t` dopo l'esecuzione delle seguenti istruzioni, se i valori inseriti attraverso la tastiera nel momento in cui vengono valutate le due espressioni `input` sono rispettivamente 1 e 2?

```
r = eval(input("Inserire un numero: "))
s = eval(input("Inserire un altro numero: "))
t = r + s
```

Commenti all'interno dei programmi

È sempre utile documentare i programmi inserendo **commenti** che indichino, per esempio, quale elaborazione viene svolta dal programma, quali sono i dati di ingresso e i risultati, e il significato delle variabili o di particolari blocchi di istruzioni.

Nei programmi Python i commenti possono essere inseriti in qualsiasi riga, preceduti dal carattere `#` ("cancelletto").

Tutti i caratteri che seguono il cancelletto, **fino al termine della stessa riga**, sono considerati commenti e vengono trascurati dall'interprete.

Negli esempi che seguono si potrà osservare l'uso dei commenti nei programmi Python.

Primi esempi di programmi Python

Un programma che acquisisce attraverso la tastiera il valore del raggio di un cerchio, e ne calcola la circonferenza.

```
raggio = eval(input("Inserire il valore del raggio: "))
pi_greco = 3.14
circonferenza = 2*pi_greco*raggio
print("La lunghezza della circonferenza è", circonferenza)
```

Il programma è disponibile nel *file* `1_circonferenza.py` nella pagina web del corso.

Primi esempi di programmi Python

Un programma che calcola l'area di un trapezio, dopo aver acquisito le lunghezze dei lati e l'altezza.
File: `2_area_trapezio.py`

```
base_minore = eval(input("Base minore del trapezio: "))
base_maggiore = eval(input("Base maggiore: "))
altezza = eval(input("Altezza: "))
area_trapezio = (base_minore + base_maggiore)*altezza/2
print("L'area è", area_trapezio)
```

L'istruzione condizionale

Nella formulazione di un algoritmo si ha spesso la necessità di esprimere la **scelta** tra due o più sequenze di istruzioni, in base al verificarsi o meno di una certa condizione durante l'**esecuzione** dello stesso algoritmo.

Esempi

- nel calcolo del valore assoluto di un numero, il risultato dipende dal segno di tale numero
- la soluzione di un'equazione di primo grado, $ax + b = 0$ è data da $-\frac{b}{a}$, **se** $a \neq 0$, **altrimenti** non è definita o non è unica

L'istruzione condizionale

In tutti i linguaggi di alto livello sono disponibili per questo scopo

- le **espressioni condizionali** (per es., $a \neq 0$), che assumono come valore **vero** oppure **falso**
- l'**istruzione condizionale**, che consente di scegliere una sequenza d'istruzioni da eseguire tra due sequenze **alternative** tra loro, in base al valore di un'espressione condizionale

Espressioni condizionali: sintassi

Il tipo più semplice di espressione condizionale consiste nel **confronto** tra il **valore** di due espressioni.

Sintassi: espressione$_1$ operatore espressione$_2$

- espressione$_1$ e espressione$_2$ 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 confronto da eseguire tra i **valori** delle due espressioni

Espressioni condizionali: 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

simbolo	**significato**
==	"uguale a" (notare il **doppio** simbolo di uguaglianza, per evitare ambiguità con l'istruzione di assegnamento)
!=	"diverso da"
<	"minore di"
<=	"minore o uguale a"
>	"maggiore di"
>=	"maggiore o uguale a"

Espressioni condizionali: esempi

Quelli che seguono sono esempi di espressioni condizionali. Si assume che a ciascuna variabile che compare in esse sia già stato assegnato un valore

- `1 < 2`
- `a + 1 != 5`
- `x >= y`
- `"macchina" < "casa"`
- `"mappa" + "mondo" == "mappamondo"`

Espressioni condizionali: semantica

Analogamente alle espressioni aritmetiche, il cui valore è un numero, e alla concatenazione di stringhe, il cui valore è una stringa, un'espressione condizionale assume il **valore logico** "vero" oppure "falso", in base all'esito del confronto.

I due valori logici vengono indicati in Python con i simboli

- `True` (vero)
- `False` (falso)

Si noti che questi simboli sono essi stessi **espressioni** lecite del linguaggio Python, proprio come i numeri e le stringhe.

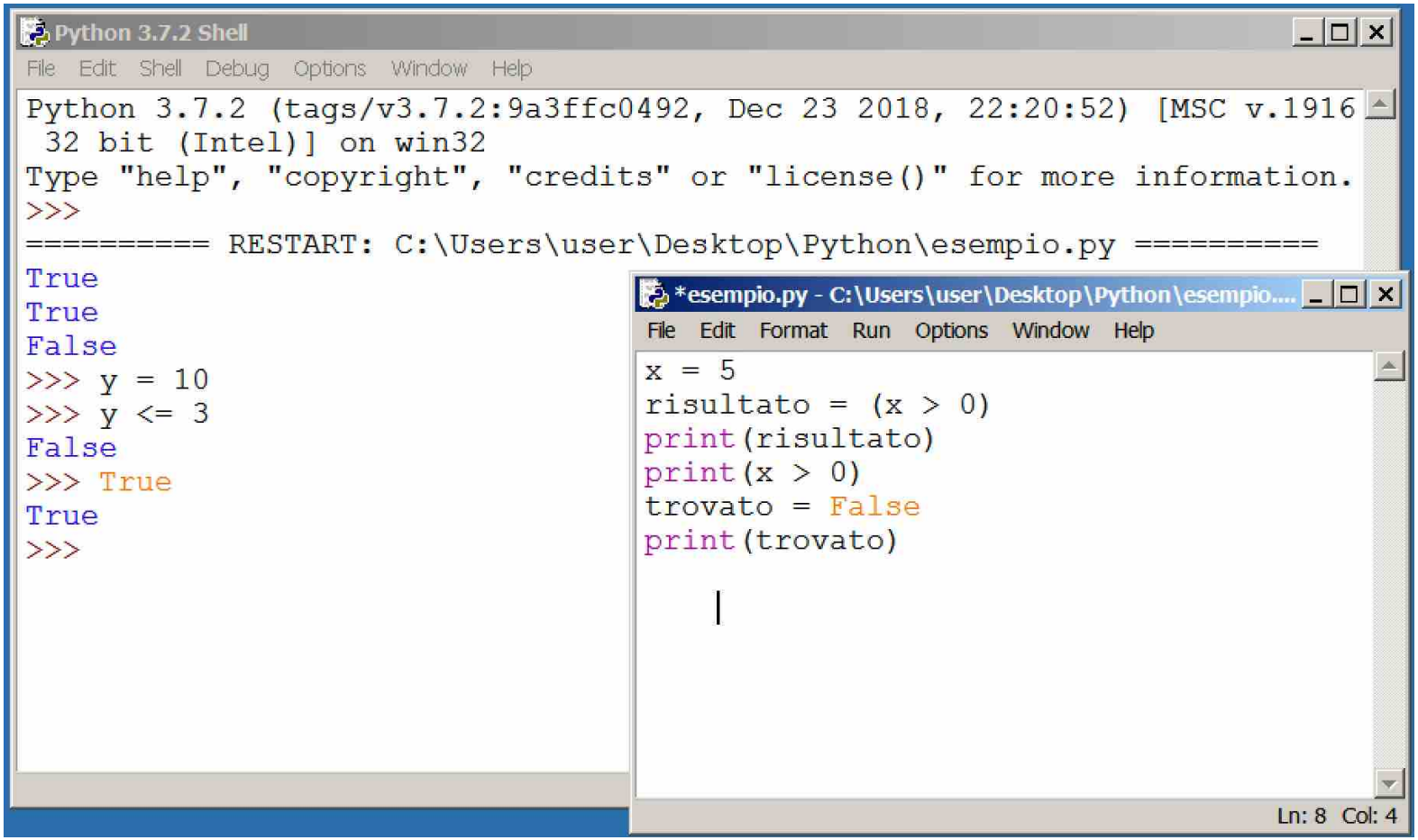
In particolare, nel caso di un confronto tra due valori di tipo stringa, gli operatori <, <=, >= e > si riferiscono all'ordinamento alfabetico, con la convenzione che le cifre **precedono** le (sono "minori" delle) lettere maiuscole, le quali a loro volta precedono le lettere minuscole.

Uso delle espressioni condizionali

Oltre che all'interno delle istruzioni condizionali (e iterative, come si vedrà più avanti), le espressioni condizionali possono essere usate all'interno di un programma Python come espressioni a sé stanti

- il loro valore può essere assegnato a una variabile; per esempio, se alla variabile `x` è già stato assegnato un valore numerico:
  ```
  risultato = (x > 0)
  ```
- il loro valore può essere stampato nella finestra della *shell* con l'istruzione `print`, per esempio:
  ```
  print(x > 0)
  ```
- possono essere scritte nella *shell*, nella quale verrà mostrato il loro valore (`True` oppure `False`)
- gli stessi simboli `True` e `False` possono essere assegnati a variabili, in quanto costituiscono espressioni, analogamente ai numeri e alle stringhe; per esempio:
  ```
  trovato = True
  ```

Espressioni condizionali: esempi



Espressioni condizionali: esempi

Si assume che a tutte le variabili che compaiono negli esempi seguenti sia già stato assegnato un valore

- `1 < 2`
  produce `True`
- `a + 1 != 5`
  produce `False` se il valore associato alla variabile `a` è 4 (oppure 4.0), altrimenti produce `True`
- `x == y`
  produce `True` se le due variabili hanno lo stesso valore, altrimenti produce `False`
- `"macchina" < "casa"`
  produce `False`

Espressioni condizionali: esempi

- `"Macchina" < "casa"`
  produce `True`
- `"mappa" + "mondo" == "mappamondo"`
  produce `True`
- `"123" < "Casa"`
  produce `True`

Espressioni condizionali: esempi

Espressioni condizionali composte

Le espressioni condizionali viste finora si dicono **semplici**, poiché consistono in un singolo confronto tra due valori.

Nel linguaggio naturale è possibile esprimere frasi complesse mediante la combinazione di frasi più semplici, attraverso **connettivi logici** come i seguenti

- la congiunzione “e”
- le congiunzioni “o”, “oppure”
- l’avverbio “non”

Analogamente, nei linguaggi di programmazione è possibile definire espressioni condizionali **composte**, mediante la combinazione di espressioni condizionali semplici.

Espressioni condizionali composte: sintassi

Sintassi:

- espr-cond$_1$ `and` espr-cond$_2$
- espr-cond$_1$ `or` espr-cond$_2$
- `not` espr-cond

dove

- espr-cond$_1$, espr-cond$_2$ e espr-cond sono espressioni condizionali semplici, o a loro volta espressioni condizionali composte
- i simboli `and`, `or` e `not` sono detti **operatori logici**, e corrispondono rispettivamente ai connettivi logici del linguaggio naturale “e”, “oppure”, “non”
- è possibile usare le **parentesi tonde** per definire l’ordine di precedenza degli operatori logici

Espressioni condizionali composte: semantica

Anche le espressioni condizionali composte assumono i valori logici `True` e `False`. Il loro valore è definito come segue

- espr-cond$_1$ `and` espr-cond$_2$
  produce `True` se **entrambe** le espressioni hanno valore `True`, altrimenti produce `False`
- espr-cond$_1$ `or` espr-cond$_2$
  produce `True` se **almeno una** delle espressioni ha valore `True`, altrimenti produce `False`
- `not` espr-cond
  produce `True` se espr-cond ha valore `False`, e viceversa

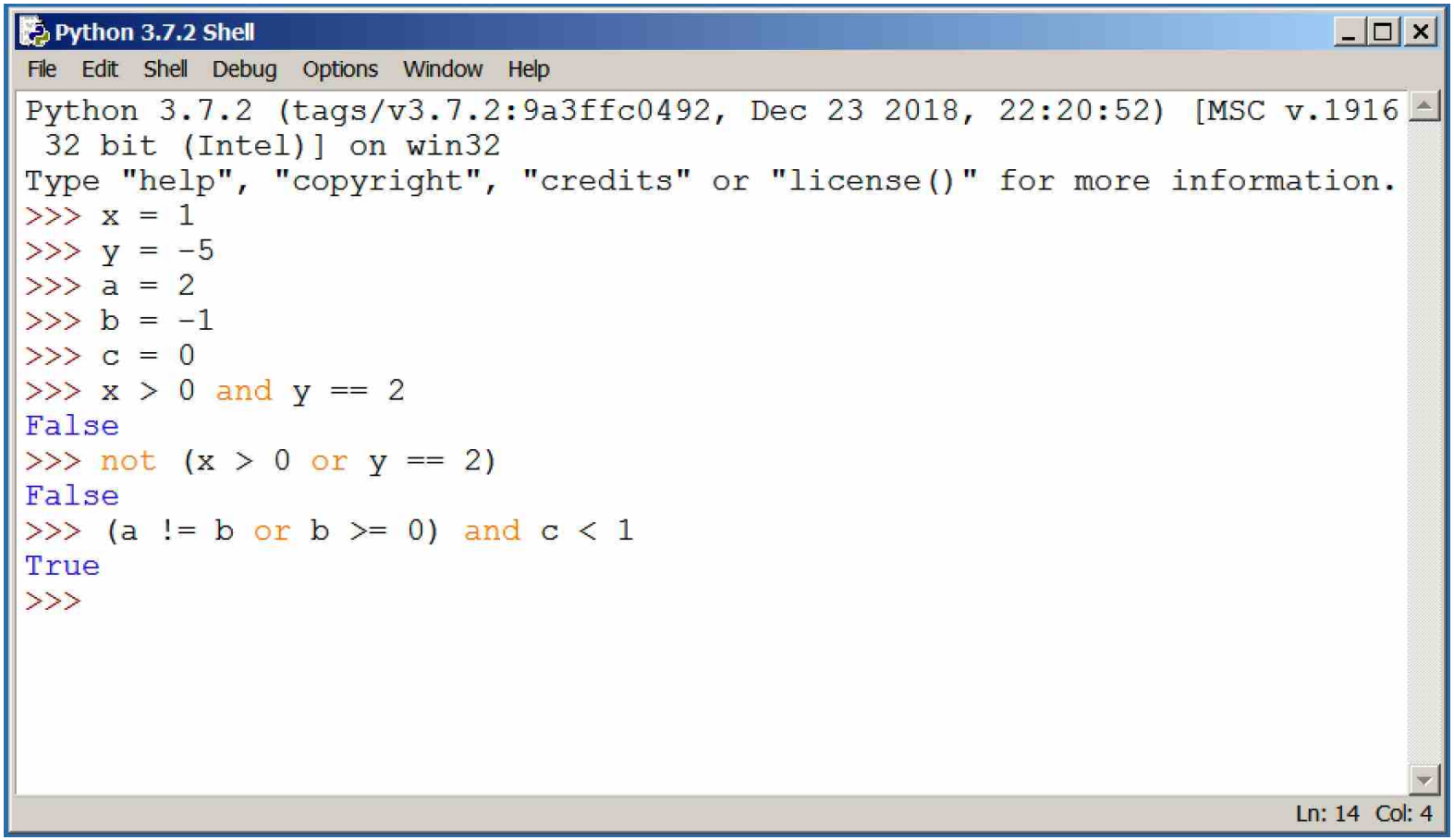
Se una qualsiasi delle espressioni condizionali componenti fosse a sua volta composta, per determinare il valore dell'espressione principale si dovrebbe prima determinare quello dell'espressione componente, con le stesse regole indicate sopra.

Espressioni condizionali composte: esempi

Come al solito, si assume che a tutte le variabili che compaiono negli esempi seguenti sia già stato assegnato un valore

- `x > 0 and y == 2`
  è vera (produce `True`) se alla variabile `x` è associato un numero positivo, **e** alla variabile `y` è associato il numero 2; altrimenti è falsa (produce `False`)
- `not (x > 0 or y == 2)`
  è vera se l'espressione tra parentesi (la stessa dell'esempio precedente) è falsa, e viceversa
- `(a != b or b >= 0) and c < 1`
  è vera se alle variabili `a` e `b` sono associati valori diversi, **oppure** se a `b` è associato un numero non negativo, **e se, contemporaneamente,** alla variabile `c` è associato un numero minore di 1; altrimenti è falsa

Espressioni condizionali: esempi



L'istruzione condizionale

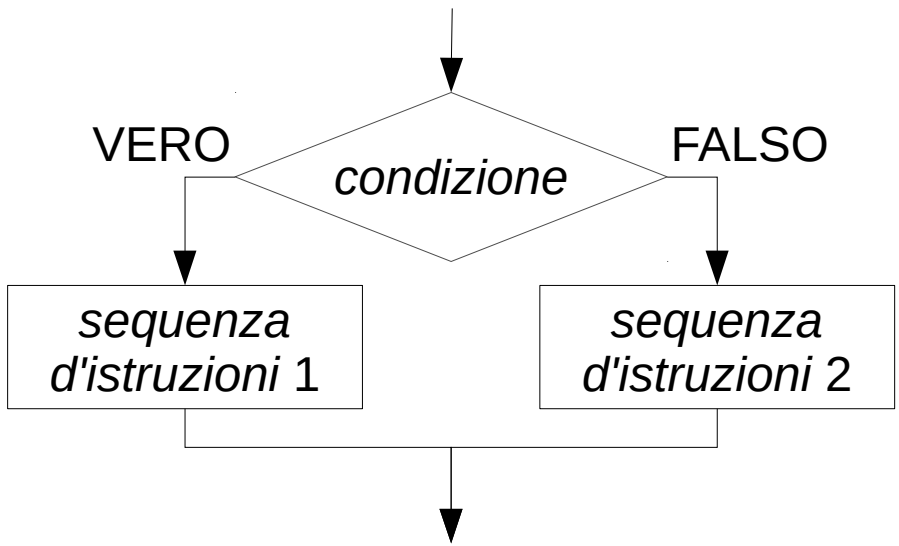
Come si è detto in precedenza, le **espressioni** condizionali sono uno dei componenti dell'**istruzione** condizionale, che consente di esprimere in un programma la **scelta** tra **due** diverse sequenze di istruzioni in base al verificarsi o meno di una certa **condizione** durante l'**esecuzione** dello stesso programma.

In linguaggio naturale, un'istruzione condizionale esprime la seguente richiesta all'esecutore di un algoritmo:

> **se** una data **condizione** è vera,
> **allora** esegui una certa sequenza d'istruzioni,
> **altrimenti** esegui un'altra sequenza d'istruzioni

L'istruzione condizionale

La semantica dell'istruzione condizionale (cioè il meccanismo della sua esecuzione) può essere rappresentata graficamente per mezzo del seguente diagramma di flusso



L'istruzione condizionale: sintassi

```
if espr-cond:
    sequenza di istruzioni 1
else:
    sequenza di istruzioni 2
```

- le parole-chiave `if` e `else` devono essere scritte **senza rientri**
- 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 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

L'istruzione condizionale: semantica

Se espressione condizionale è vera (cioè, se il suo valore è `True`), viene eseguita la sequenza di istruzioni 1 (le sue istruzioni vengono eseguite nello stesso ordine nel quale sono scritte).

Se invece espressione condizionale è falsa, viene eseguita la sequenza di istruzioni 2.

Si noti che **solo una** delle due sequenze di istruzioni viene eseguita.

L'istruzione condizionale: esempio

Si assuma che alla variabile `x` sia già stato assegnato un valore:

```
if x > 0:
   print("La condizione è vera.")
   z = x + 1
else:
   print("La condizione è falsa.")
```

Se il valore associato alla variabile `x` (nel momento in cui l'istruzione condizionale viene **eseguita**) è un numero positivo, viene prima mostrato (nella *shell*) il messaggio

```
La condizione è vera.
```

poi viene assegnato alla variabile `z` il valore dell'espressione `x + 1`.
Altrimenti (se l'espressione `x > 0` è falsa) viene mostrato il messaggio

```
La condizione è falsa.
```

NOTA: dato che l'istruzione condizionale è composta da più righe, per non confondersi con i rientri si consiglia di scrivere gli esempi che seguono in una finestra dell'*editor* (eseguendoli come programmi) piuttosto che nella *shell*.

L'istruzione condizionale: una variante

L'istruzione condizionale può essere usata anche quando non è prevista l'esecuzione di istruzioni nel caso in cui la condizione sia **falsa** (in altre parole, la parte introdotta da `else` può essere assente).

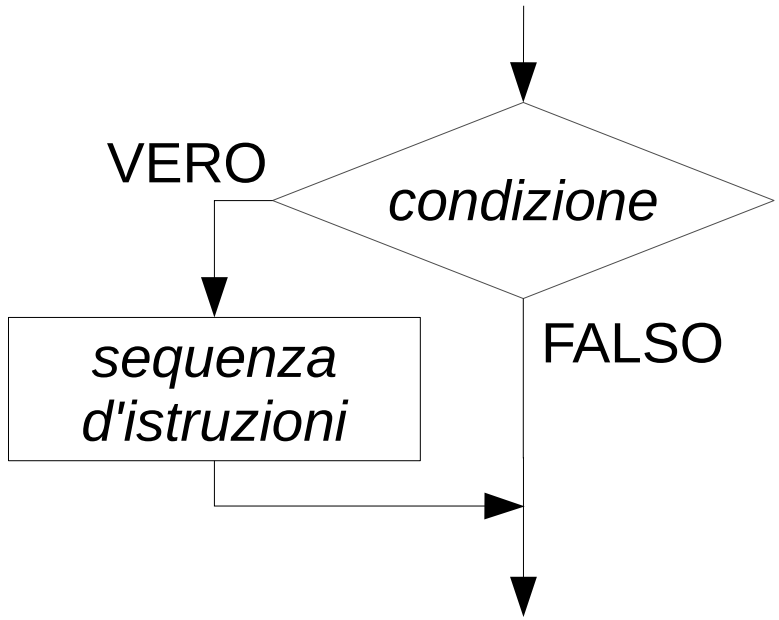
Sintassi:

```
if espr-cond:
    sequenza di istruzioni
```

Semantica: se espr-cond è vera, allora viene eseguita la sequenza di istruzioni; in caso contrario, si passa alla (eventuale) istruzione **successiva** a quella condizionale.

L'istruzione condizionale: una variante

Il diagramma di flusso corrispondente è il seguente



L'istruzione condizionale: esempio

Si assuma che alla variabile `x` sia già stato assegnato un valore:

```
if x > 0:
    print("La condizione è vera.")
    z = x + 1
```

Se il valore associato alla variabile `x` (nel momento in cui l'istruzione condizionale viene **eseguita**) è un numero positivo, viene prima mostrato (nella *shell*) il messaggio

```
La condizione è vera.
```

poi viene assegnato alla variabile `z` il valore dell'espressione `x + 1`.

In caso contrario non viene eseguita nessuna istruzione.

Ancora sulla sintassi dell'istruzione condizionale

I **rientri** (ovvero gli spazi all'inizio di una riga) 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** rispetto a essa.

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 chiamate di `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** chiamata di `print` dopo `if` è scritta con un rientro, e quindi solo essa fa parte dell'istruzione condizionale; la seconda verrà invece eseguita **dopo** l'istruzione condizionale, indipendentemente dal valore della condizione `x > 0`.

Istruzione condizionale: esercizio

Determinare che cosa viene stampato nella *shell* dalla sequenza di istruzioni mostrata in basso, nel caso in cui **prima** della loro esecuzione il valore associato alle variabili `x` e `y` sia 1, e nel caso in cui il valore associato a `x` sia -1 e quello associato a `y` ancora 1.

```
if x > 0:
   print("A")
   y = 2
if y <= 2:
   print("B")
   print("C")
else:
   print("D")
print("E")
```

Istruzioni condizionali *nidificate*

Un'istruzione condizionale può contenere al suo interno istruzioni qualsiasi, e quindi anche altre istruzioni condizionali. Si parla in questo caso di istruzioni **nidificate** (o **annidate**).

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

Un'istruzione condizionale nidificata all'interno di un'altra si scrive con la stessa sintassi mostrata sopra; questo implica che

- le parole-chiave `if` e (se presente) `else` devono essere scritte con un **rientro** rispetto a quelle dell'istruzione condizionale che le contiene
- le sequenze di istruzioni che seguono `if` e `else` devono essere scritte con un **ulteriore rientro**

Istruzioni condizionali nidificate: esempio

Nell'esempio in basso l'istruzione condizionale

```
if y == 1: ...
```

è nidificata all'interno di un'altra istruzione condizionale:

```
if x > 0: ...
```

(si assume che alle variabili `x` e `y` sia già stato assegnato un valore).
Si noti che entrambe contengono anche la parte `else`.

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

Istruzioni condizionali nidificate: esempio

Per capire meglio una sequenza di istruzioni come quella mostrata in precedenza è utile **eseguirla** passo dopo passo, così come verrebbe effettivamente eseguita dal calcolatore.

A questo scopo bisogna individuare prima di tutto le due sequenze di istruzioni che vengono eseguite nel caso in cui l'espressione condizionale della **prima** istruzione `if` (quella "esterna") sia vera, e nel caso in cui tale espressione sia falsa.

Questa informazione è sempre fornita **esclusivamente** dai **rientri** all'inizio di ciascuna riga, secondo la sintassi già descritta.

Istruzioni condizionali nidificate: esempio

Le due sequenze di istruzioni contenute nella prima istruzione `if` sono evidenziate in basso in magenta (per il caso in cui `x > 0` è vera) e in blu (per il caso in cui `x > 0` è falsa).

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

Istruzioni condizionali nidificate: esempio

In particolare, la sequenza corrispondente al caso in cui `x > 0` sia vera è composta da **tre** istruzioni

- l'istruzione `print("A")`
- l'istruzione condizionale (nidificata) `if y == 1: ...`
- l'istruzione `print("D")`

Si noti che l'istruzione `print("D")` **non** fa parte dell'istruzione condizionale `if y == 1: ...`, e quindi verrà eseguita (nel caso in cui `x > 0` sia vera) **dopo** di essa, indipendentemente dal valore di verità della condizione `y == 1`.

Istruzioni condizionali nidificate: esempio

In conclusione

- se `x > 0` è vera
  - prima viene stampato `A`
  - poi, se `y == 1` è vera viene stampato `B`, altrimenti viene stampato `C`
  - poi viene stampato `D`
- se invece `x > 0` è falsa, viene solo stampato `E`

Istruzioni condizionali nidificate: esercizio

Determinare ciò che viene stampato nella *shell* dall'esecuzione delle istruzioni dell'esempio precedente, per ciascuno dei seguenti valori delle variabili `x` e `y`

- `x = -1, y = -1`
- `x = 2, y = 1`
- `x = -2, y = 1`
- `x = 3, y = 2`

Esempi di programmi contenenti istruzioni condizionali

Un programma che calcola il valore assoluto di un numero acquisito attraverso la tastiera.
File: `3_valore_assoluto_1.py`

```
n = eval(input("Inserire un numero: "))
if n > 0:
    print("Valore assoluto:", n)
else:
    print("Valore assoluto:", -n)
```

Esempi di programmi contenenti istruzioni condizionali

Una versione alternativa, nella quale si usa un'istruzione condizionale senza la parte `else`.
File: `4_valore_assoluto_2.py`

```
n = eval(input("Inserire un numero: "))
if n < 0:
    n = -n
print("Valore assoluto:", n)
```

Esempi di programmi contenenti istruzioni condizionali

Un programma che acquisisce i coefficienti di un'equazione di primo grado, $ax + b = 0$, e ne calcola la soluzione, se questa esiste ed è unica, altrimenti stampa un messaggio opportuno.
File: `5_eq_primo_grado.py`

```
a = eval(input("Inserire il coefficiente a: "))
b = eval(input("Inserire il coefficiente b: "))
if a != 0:
    print("La soluzione è", - b/a)
else:
    print("La soluzione non esiste o non è unica.")
```

Esempi di programmi contenenti istruzioni condizionali

Un programma che acquisisce i coefficienti di un'equazione di secondo grado, $ax^2 + bx + c = 0$, e determina se le radici siano reali oppure complesse, stampando un opportuno messaggio.
File: `6_eq_secondo_grado_1.py`

```
a = eval(input("Inserire il coefficiente a: "))
b = eval(input("Inserire il coefficiente b: "))
c = eval(input("Inserire il coefficiente c: "))
delta = b**2 - 4*a*c
if delta >= 0:
    print("Le radici sono reali.")
else:
    print("Le radici sono complesse.")
```

Esempi di programmi contenenti istruzioni condizionali

Una variante dello stesso programma, che stampa anche i valori delle radici. *File*: `7_eq_secondo_grado_2.py`

```
a = eval(input("Inserire il coefficiente a: "))
b = eval(input("Inserire il coefficiente b: "))
c = eval(input("Inserire il coefficiente c: "))
delta = b**2 - 4*a*c
if delta >= 0:
    print("Le radici sono reali.")
    print("x1 =", (-b + delta**0.5)/(2*a))
    print("x2 =", (-b - delta**0.5)/(2*a))
else:
    print("Le radici sono complesse.")
    print("x1 =", -b/(2*a), ((-delta)**0.5)/(2*a))
    print("x2 =", -b/(2*a), - ((-delta)**0.5)/(2*a))
```

Esempi di programmi contenenti istruzioni condizionali

Un'altra variante, che determina se le radici siano reali e distinte, reali e coincidenti, oppure complesse coniugate. In questo esempio vengono usate tre istruzioni condizionali in sequenza (non nidificate), corrispondenti a tre condizioni **mutuamente esclusive**: quindi una sola delle istruzioni corrispondenti verrà eseguita.
File: `8_eq_secondo_grado_3.py`

```
a = eval(input("Inserire il coefficiente a: "))
b = eval(input("Inserire il coefficiente b: "))
c = eval(input("Inserire il coefficiente c: "))
delta = b**2 - 4*a*c
if delta > 0:
    print("Le radici sono reali e distinte.")
if delta == 0:
    print("Le radici sono reali e coincidenti.")
if delta < 0:
    print("Le radici sono complesse coniugate.")
```

Esempi di programmi contenenti istruzioni condizionali

Una quarta variante, nella quale si usano istruzioni condizionali nidificate. *File*: `9_eq_secondo_grado_4.py`

```
a = eval(input("Inserire il coefficiente a: "))
b = eval(input("Inserire il coefficiente b: "))
c = eval(input("Inserire il coefficiente c: "))
delta = b**2 - 4*a*c
if delta > 0:
    print("Le radici sono reali e distinte.")
else:
    if delta == 0:
        print("Le radici sono reali e coincidenti.")
    else:
        print("Le radici sono complesse coniugate.")
```

Esempi di programmi contenenti istruzioni condizionali

Il programma mostrato di seguito acquisisce tre numeri, e determina se essi possano rappresentare le lunghezze dei lati di un triangolo (cioè se siano tutti positivi, e se ciascuno sia minore della somma degli altri due); in caso affermativo determina se si tratti di un triangolo equilatero, isoscele o scaleno, altrimenti stampa un messaggio opportuno.

Si noti che in linguaggio Python è possibile suddividere **una** istruzione in più righe, inserendo il carattere `\` (*backslash*) nel punto in cui si interrompe una riga. Questa possibilità è stata sfruttata per suddividere in due righe l'espressione condizionale della prima istruzione `if`.

Nella prosecuzione di una riga si può inserire un rientro qualsiasi (anche nessuno). Per garantire la leggibilità del programma è però buona norma usare un rientro coerente con il contenuto della riga precedente.

Esempi di programmi contenenti istruzioni condizionali

File: `10_triangoli_1.py`

```
a = eval(input("Inserire il primo numero: "))
b = eval(input("Inserire il secondo numero: "))
c = eval(input("Inserire il terzo numero: "))
if a > 0 and b > 0 and c > 0 and \
   a < b + c and b < a + c and c < a + b:
    if a == b and b == c:
        print("Equilatero")
    else:
        if a == b or b == c or a == c:
            print("Isoscele")
        else:
            print("Scaleno")
else:
    print("Non rappresentano i lati di un triangolo.")
```

Esempi di programmi contenenti istruzioni condizionali

Di seguito si mostra una seconda versione dello stesso programma, con una diversa espressione condizionale nella prima istruzione `if` (tale espressione verifica se i tre numeri **non** corrispondano alle lunghezze dei lati di un triangolo).

Questo consente di spostare nella parte `else` della stessa istruzione la determinazione del tipo di triangolo, rendendo il programma più leggibile.

Esempi di programmi contenenti istruzioni condizionali

File: `11_triangoli_2.py`

```
a = eval(input("Inserire un numero: "))
b = eval(input("Inserire un altro numero: "))
c = eval(input("Inserire l'ultimo numero: "))
if not (a > 0 and b > 0 and c > 0 and \
        a < b + c and b < a + c and c < a + b):
    print("Non rappresentano i lati di un triangolo.")
else:
    if a == b and b == c:
        print("Equilatero")
    else:
        if a == b or b == c or a == c:
            print("Isoscele")
        else:
            print("Scaleno")
```

L'istruzione condizionale: una seconda variante

Il programma nell'esempio precedente prevede quattro possibili condizioni, a ciascuna delle quali corrisponde una diversa sequenza d'istruzioni. Ciò è stato espresso mediante una "cascata" di istruzioni condizionali `if...else...` nidificate.

Per poter esprimere in modo più semplice una sequenza di alternative tra più di due condizioni, l'istruzione condizionale del linguaggio Python prevede una variante con la seguente sintassi:

```
if espr-cond-1:
    sequenza di istruzioni 1
elif espr-cond-2:
    sequenza di istruzioni 2
...
else:
    sequenza di istruzioni alternativa
```

Si noti l'assenza di rientri nelle righe che iniziano con `elif`.

L'istruzione condizionale: una seconda variante

Quella precedente è una **singola** istruzione condizionale, che può contenere un numero **qualsiasi** di parti `elif` (e di corrispondenti condizioni e sequenze d'istruzioni); inoltre può anche **non** contenere la parte `else` finale.

La semantica di questa istruzione prevede la valutazione in sequenza delle varie espressioni condizionali

- non appena una di esse risulta vera, viene eseguita la corrispondente sequenza d'istruzioni e l'esecuzione dell'intera istruzione condizionale termina (quindi le espressioni condizionali successive **non** vengono valutate).
- se **nessuna** delle espressioni condizionali che seguono `if` e `elif` risulta vera, viene eseguita la sequenza d'istruzioni della parte `else` (se questa è presente)

Esempio

Il programma visto in precedenza può essere riscritto come segue (si veda il *file* `11_triangoli_3.py`)

```
a = eval(input("Inserire un numero: "))
b = eval(input("Inserire un altro numero: "))
c = eval(input("Inserire l'ultimo numero: "))
if not (a > 0 and b > 0 and c > 0 and \
        a < b + c and b < a + c and c < a + b):
    print("Non rappresentano i lati di un triangolo.")
elif a == b and b == c:
    print("Equilatero")
elif a == b or b == c or a == c:
    print("Isoscele")
else:
    print("Scaleno")
```

L'istruzione iterativa

In molti algoritmi è necessario **ripetere** più volte una stessa sequenza di operazioni.

Per esempio, il procedimento per la determinazione delle cifre di un numero N in una base b richiede di calcolare ripetutamente il quoziente e il resto della divisione per b di N (nel primo passo) o dell'ultimo quoziente ottenuto (nei passi successivi), fino a ottenere un quoziente pari a zero.

In molti linguaggi di alto livello è presente a questo scopo l'**istruzione iterativa**, che consente di esprimere la seguente richiesta all'esecutore di un algoritmo:

> **finché** una data **condizione** è vera,
> **esegui** una certa sequenza di istruzioni

L'istruzione iterativa: sintassi

La sintassi è simile a quella dell'istruzione condizionale:

```
while espr-cond:
    sequenza di istruzioni
```

- la parola chiave `while` deve essere scritta **senza rientri**
- espr-cond è un'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

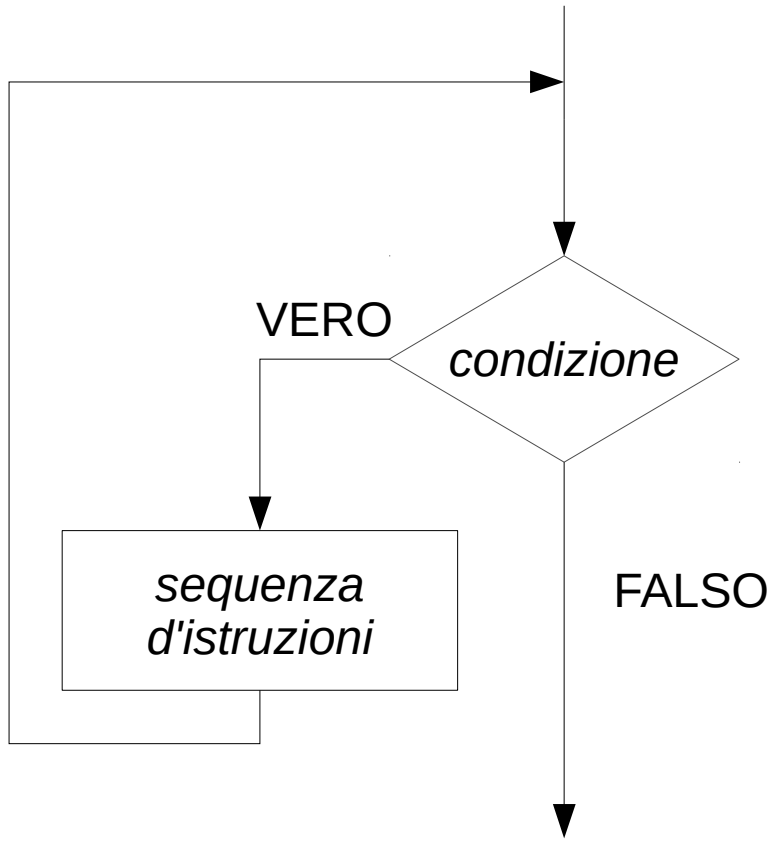
L'istruzione iterativa: semantica

Un'istruzione iterativa viene eseguita **ripetendo** ciclicamente i seguenti passi

1. viene valutata espr-cond
2. se espr-cond è vera, si esegue la sequenza di istruzioni, e si **riprende** l'esecuzione dal punto **??**; se invece espr-cond è falsa l'esecuzione dell'istruzione iterativa termina, e si passa all'eventuale istruzione **successiva** a quella iterativa

L'istruzione iterativa: semantica

Lo schema di esecuzione dell'istruzione iterativa corrisponde al seguente diagramma di flusso



L'istruzione iterativa: esempi

Come per l'istruzione condizionale, si consiglia di scrivere i programmi mostrati negli esempi seguenti in una finestra dell'*editor* invece che nella *shell*.

L'istruzione iterativa mostrata nell'esempio in basso richiede la ripetizione di due istruzioni finché la condizione $1 > 2$ risulta vera: stampare nella *shell* la stringa `Python` e assegnare alla variabile `x` il valore 7.

```
while 1 > 2:
    print("Python")
    x = 7
```

Dato che la condizione $1 > 2$ è falsa, l'esecuzione dell'istruzione `while` termina subito dopo la verifica di tale condizione, senza che le due istruzioni al suo interno vengano mai eseguite.

L'istruzione iterativa: esempi

Un esempio simile al precedente, con una diversa espressione condizionale:

```
while 1 < 2:
    print("Python")
    x = 7
```

Dato che la condizione risulta **sempre** vera, le due istruzioni all'interno dell'istruzione `while` vengono ripetute (in teoria) **infinite** volte. Come effetto visibile, nella *shell* sarà ripetutamente stampata la stringa `"Python"`.

In casi come questo è possibile interrompere l'esecuzione di un programma scegliendo la voce `Interrupt Execution` dal *menu* `Shell` dell'ambiente `IDLE`, oppure premendo i tasti `CTRL` + `C`

L'istruzione iterativa: esempi

In questo esempio (disponibile anche nel *file* `12_while_1.py`), nell'espressione condizione compare una variabile alla quale viene assegnato un valore **prima** dell'istruzione iterativa; il suo valore viene poi modificato **durante** l'esecuzione di tale istruzione.

In questo modo il valore della condizione può essere vero all'inizio della **prima** iterazione (e di alcune delle iterazioni successive), e può successivamente diventare falso, consentendo la conclusione dell'istruzione iterativa dopo un numero **finito** di ripetizioni.

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

L'istruzione iterativa: esempi

Per comprendere meglio il meccanismo di eseczione di un'istruzione iterativa, si suggerisce di provare a eseguirla passo dopo passo con carta e matita, tenendo traccia in ogni istante di

- quale delle istruzioni della sequenza da ripetere sia in esecuzione
- quali siano i valori delle variabili
- che cosa viene stampato nella *shell*

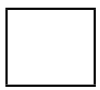
Questo metodo può essere usato più in generale per comprendere l'esecuzione di un programma qualsiasi.

L'istruzione iterativa: esempi

Di seguito si schematizza l'esecuzione del programma seguendo il metodo suggerito in precedenza

- a sinistra si riporta il programma e si evidenzia (con il colore magenta) l'istruzione in esecuzione
- al centro
  - in alto si mostra il valore associato alla variabile `k` **dopo** l'esecuzione dell'istruzione evidenziata
  - in basso si mostra (in blu) ciò che compare nella *shell*

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k 

L'istruzione iterativa: esempi

Il programma è composto da **due istruzioni**: un'istruzione di assegnamento e un'istruzione iterativa (che a sua volta contiene altre istruzioni).

La prima istruzione che viene eseguita è l'assegnamento `k = 1`:

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 1

L'istruzione iterativa: esempi

Inizia quindi l'esecuzione dell'istruzione iterativa. Viene prima di tutto valutata l'espressione condizionale, che risulta vera; di conseguenza, verranno successivamente eseguite le istruzioni all'interno dell'istruzione iterativa.

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 1 |

L'istruzione iterativa: esempi

La prima di tali istruzioni (più precisamente, si tratta di una chiamata di funzione) causa la stampa della stringa `Buongiorno` nella *shell*

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 1 |

```
Buongiorno
```

L'istruzione iterativa: esempi

La seconda istruzione modifica il valore associato alla variabile `k`, incrementando di una unità il valore `attuale`

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 2 |

```
Buongiorno
```

L'istruzione iterativa: esempi

L'istruzione `k = k + 1` appena eseguita è l'ultima della sequenza all'interno dell'istruzione `while` (come si vede dai rientri).
L'esecuzione dell'istruzione `while` procede perciò ripartendo dalla valutazione dell'espressione condizionale, che risulta di nuovo vera.
Le due istruzioni al suo interno verranno quindi ripetute una seconda volta.

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 2 |

```
Buongiorno
```

L'istruzione iterativa: esempi

Si esegue prima la funzione `print`, che stampa nella *shell* per la **seconda** volta la stringa `Buongiorno`

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 2 |

```
Buongiorno
Buongiorno
```

L'istruzione iterativa: esempi

Si esegue poi l'istruzione di assegnamento, che incrementa il valore associato a `k`

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 3

```
Buongiorno
Buongiorno
```

L'istruzione iterativa: esempi

Si riparte quindi dalla valutazione dell'espressione condizionale dell'istruzione `while`, che è ancora vera: le istruzioni al suo interno verranno quindi ripetute per la **terza** volta

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 3

```
Buongiorno
Buongiorno
```

L'istruzione iterativa: esempi

Il messaggio `Buongiorno` viene stampato per la terza volta nella *shell*

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 3

```
Buongiorno
Buongiorno
Buongiorno
```

L'istruzione iterativa: esempi

Il valore associato a `k` viene incrementato per la terza volta, diventando ora 4

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 4 |

```
Buongiorno
Buongiorno
Buongiorno
```

L'istruzione iterativa: esempi

Si valuta quindi l'espressione condizionale, che questa volta risulta **falsa**: l'esecuzione dell'istruzione `while` a questo punto termina (senza che vengano di nuovo eseguite le istruzioni al suo interno), e non essendoci nessuna istruzione **successiva** (scritta cioè con lo stesso rientro della parola-chiave `while`) termina anche l'esecuzione del programma

```
k = 1
while k <= 3:
    print("Buongiorno")
    k = k + 1
```

k | 4

```
Buongiorno
Buongiorno
Buongiorno
```

L'istruzione iterativa: esempi

Quello precedente è un esempio di istruzione iterativa nella quale il **numero di ripetizioni** è **noto** nel momento in cui si **scrive** il programma (in questo esempio, **tre** ripetizioni).

In questo caso si usa comunemente una variabile (nell'esempio, la variabile `k`) alla quale **prima** dell'istruzione iterativa si assegna un valore scelto dal programmatore (di norma, 1); tale valore viene poi **incrementato** di **una** unità **in ogni iterazione**; l'espressione condizionale verifica che il valore associato a tale variabile sia minore o uguale al numero desiderato di ripetizioni.

Poiché la funzione di tali variabili è di tenere traccia del numero di iterazioni, ovvero "contare" il numero di iterazioni, esse vengono comunemente indicate con il termine "**contatori**".

L'istruzione iterativa: esempi

Il programma seguente (disponibile nel *file* `13_while_2.py`) è simile a quello appena visto, ma questa volta il numero di iterazioni dipende dal valore associato a `n`, che viene acquisito attraverso la tastiera all'inizio dello stesso programma. Il numero di iterazioni non è quindi noto nel momento in cui si **scrive** il programma.

Si suggerisce di eseguire anche questo programma con carta e matita (prima di farlo eseguire dal calcolatore), scegliendo arbitrariamente il valore che verrà restituito dall'espressione `input`.

```
n = eval(input("Inserire un numero: "))
k = 1
while k <= n:
    print("Buongiorno")
    k = k + 1
```

L'istruzione iterativa: esempi

Il programma in basso (disponibile nel *file* `14_while_3.py`) acquisisce una sequenza di numeri e stampa il quadrato di ciascuno di essi, arrestandosi quando viene acquisito un numero pari a zero; al termine viene stampato il messaggio `Fine`.

Si noti che la chiamata `print("Fine.")` si trova **al di fuori** dell'istruzione `while` (poiché è scritta **senza rientri**), e quindi viene eseguita **una sola volta**, **dopo** che l'esecuzione dell'istruzione `while` è terminata.

```
n = eval(input("Inserire un numero: "))
while n != 0:
    print("Il suo quadrato è", n**2))
    n = eval(input("Inserire un altro numero: "))
print("Fine.")
```

L'istruzione iterativa: esempi

Il programma mostrato di seguito (disponibile nel *file* `15_while_4.py`) acquisisce un numero (che si richiede essere un intero) e stampa a ritroso, a partire da questo, i numeri interi fino allo zero **incluso**. Si noti che se il valore acquisito è negativo non viene stampato nessun numero, poiché in questo caso l'espressione condizionale dell'istruzione `while` risulta subito falsa.

```
n = eval(input("Inserire un numero intero: "))
while n >= 0:
    print(n)
    n = n - 1
```

L'istruzione iterativa: esercizi

Eseguire con carta e matita il programma seguente (disponibile nel *file* `16_while_5.py`), assumendo che il valore acquisito attraverso l'espressione `input` sia 3, e determinare che cosa viene stampato nella *shell*

```
n = eval(input("Inserire un numero: "))
x = 0
k = 1
while k <= n:
    x = x + 1/k
    k = k + 1
print(x)
```

L'istruzione iterativa: esercizi

Eseguire con carta e matita il programma seguente (disponibile nel *file* `17_while_6.py`), assumendo che il primo valore acquisito sia 3 e che i successivi siano 4, -3 e 6, e determinare che cosa viene stampato nella *shell*

```
n = eval(input("Inserire un numero: "))
a = 0
z = 1
while z <= n:
    x = eval(input("Inserire un numero: "))
    a = a + x
    z = z + 1
print("Il risultato è", a)
```

L'istruzione iterativa: esercizi

Il programma mostrato di seguito (disponibile nel *file* `18_while_7.py`) contiene un'istruzione condizionale nidificata all'interno di un'istruzione iterativa.

Osservando i rientri si deduce che l'istruzione iterativa contiene una sequenza di **tre** istruzioni: un assegnamento (`q = ...`), un'istruzione condizionale **senza** la parte `else`, e un altro assegnamento (`i = i + 1`).

L'istruzione condizionale contiene a sua volta **una sola istruzione** (l'assegnamento `b = b + 1`).

Si può infine osservare che la chiamata di `print` non fa parte dell'istruzione iterativa, e verrà quindi eseguita una sola volta **dopo** tale istruzione.

L'istruzione iterativa: esercizi

```
n = eval(input("Inserire un numero: "))
b = 0
i = 1
while i <= n:
    q = eval(input("Inserire un numero: "))
    if q > 0:
        b = b + 1
    i = i + 1
print("Risultato:", b)
```

Eseguire questo programma con carta e penna, assumendo che il primo valore acquisito sia 4 e che i successivi siano 2, -8, -3 e 5, e determinare che cosa viene stampato nella *shell*

L'istruzione iterativa: esercizi

Quest'ultimo esercizio (il programma è disponibile nel *file* `19_while_8.py`) contiene un esempio di istruzione iterativa nidificata all'interno di un'altra istruzione iterativa.

Si osservi che la prima istruzione iterativa (`while a < 4:...`) contiene una sequenza di **tre** istruzioni: un assegnamento (`b = 1`), un'istruzione iterativa nidificata (`while b < 3:...`) e un altro assegnamento (`a = a + 1`).

L'istruzione iterativa nidificata contiene a sua volta una sequenza composta da **due** istruzioni: una chiamata della funzione `print` e un assegnamento (`b = b + 1`).

L'istruzione iterativa: esercizi

```
a = 1
while a < 4:
    b = 1
    while b < 3:
        print(a, b)
        b = b + 1
    a = a + 1
```

Eseguire questo programma con carta e penna, e determinare che cosa viene stampato nella *shell*

Esempi di programmi Python

Di seguito si mostrano esempi di programmi che eseguono operazioni di senso compiuto e non banali, e che fanno uso degli elementi del linguaggio Python visti in precedenza.

Si ricorda che per poter comprendere le operazioni svolte da un programma è **indispensabile** aver compreso

- la sintassi e il meccanismo di esecuzione delle **singole istruzioni** (assegnamento, condizionale, iterativa) e delle **chiamate di funzione**
- la sintassi e il meccanismo di valutazione delle **espressioni**

Struttura dei programmi

In generale un programma prevede

- l'acquisizione dei dati da elaborare (attraverso la tastiera con la funzione `input`, o con altri meccanismi che saranno presentati più avanti) e il loro assegnamento a opportune variabili
- l'elaborazione dei dati d'ingresso e degli eventuali risultati intermedi fino a ottenere i risultati desiderati (i risultati intermedi dovranno essere memorizzati in opportune variabili)
- la stampa dei risultati sullo schermo per mezzo della funzione `print` (oppure il loro invio ad altri dispositivi periferici, come si vedrà più avanti)

Esempi di programmi Python

Il programma seguente (disponibile nel *file* `20_stampa_numeri_pari.py`) stampa tutti i numeri pari compresi tra 1 e un numero acquisito attraverso la tastiera

```
massimo = eval(input("Estremo superiore:  "))
n = 2
print("I numeri pari tra 2 e", massimo, "sono:")
while n <= massimo:
    print(n)
    n = n + 2
```

Esempi di programmi Python

Il programma mostrato di seguito calcola il valore più piccolo in una sequenza di numeri di lunghezza *qualsiasi*, acquisita attraverso la tastiera. Anche la dimensione della sequenza è un dato d'ingresso, ed è il primo a essere acquisito. Il programma è disponibile nel *file* `21_minimo.py`

Il procedimento consiste nell'acquisire i valori della sequenza per mezzo di un'istruzione iterativa e nel tener traccia, istante per istante, del valore più piccolo tra quelli già acquisiti. Tale valore viene memorizzato nella variabile `minimo`. Il valore associato a tale variabile viene aggiornato ogni qual volta se ne incontra uno più piccolo.

Non è difficile rendersi conto che questo algoritmo richiede che il valore iniziale della variabile `minimo` debba essere pari al **primo** valore della sequenza (oppure al valore $+\infty$, che però non può essere rappresentato in linguaggio Python).

Esempi di programmi Python

```
n = eval(input("Lunghezza della sequenza? "))
x = eval(input("Primo valore: "))
minimo = x
k = 2
while k <= n:
    x = eval(input("Prossimo valore: "))
    if x < minimo:
        minimo = x
    k = k + 1
print("Il valore più piccolo è", minimo)
```

Esempi di programmi Python

Il programma seguente (disponibile nel *file* `22_somma.py`) acquisisce attraverso la tastiera una sequenza di numeri di lunghezza *qualsiasi* e ne calcola la somma (come nell'esempio precedente, anche la lunghezza della sequenza è un dato d'ingresso). Il risultato viene calcolato memorizzando nella variabile `somma` il valore della somma **parziale** dei numeri della sequenza, durante la loro acquisizione. Per questo motivo il valore iniziale di tale variabile deve essere **zero**.

```
n = eval(input("Quanti numeri si vogliono sommare? "))
somma = 0
k = 1
while k <= n:
    x = eval(input("Prossimo valore: "))
    somma = x + somma
    k = k + 1
print("La somma è", somma)
```

Esempi di programmi Python

Il programma seguente (disponibile nel *file* `23_serie_armonica.py`) calcola la somma dei primi m termini della serie armonica, per un dato valore di m:

$$\sum_{k=1}^{m} \frac{1}{k}$$

L'algoritmo per il calcolo della somma è lo stesso dell'esempio precedente; l'unica differenza è che i valori da sommare devono essere **calcolati**, non acquisiti attraverso la tastiera.

```
m = eval(input("Numero di termini della serie armonica: "))
serie = 0
k = 1
while k <= m:
    serie = serie + 1/k
    k = k + 1
print("Somma dei primi", m, "termini:", serie)
```

Esempi di programmi Python

Il programma seguente (disponibile nel *file* `24_fattoriale.py`) calcola il fattoriale di un dato numero naturale n, definito come segue:

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n, \quad \text{se } n > 0,$$
$$n! = 1, \quad \text{se } n = 0$$

Il procedimento di calcolo è analogo a quello per la somma di una sequenza di numeri: il risultato viene calcolato memorizzando nella variabile `fatt` il valore del prodotto **parziale** dei numeri $1, 2, \ldots, n$, per mezzo di un'istruzione iterativa.

```
n = eval(input("Inserire un numero naturale: "))
fatt = 1
k = 2
while k <= n:
    fatt = fatt*k
    k = k + 1
print("Il fattoriale di", n, "è", fatt)
```

Esempi di programmi Python

Un programma (disponibile nel *file* `25_base_due.py`) che calcola le cifre della rappresentazione in base due di un numero naturale, usando il noto procedimento delle divisioni successive per due. Notare che le cifre vengono stampate nell'ordine in cui vengono calcolate (dalla meno significativa alla più significativa).

```
n = eval(input("Numero naturale da esprimere in base due: "))
if n == 0:
    print(0)
while n != 0:
    print(n % 2)
    n = n//2
```

Esempi di programmi Python

Il programma mostrato di seguito (disponibile nel *file* `26_MCD_1.py`) calcola il massimo comun divisore (MCD) di due numeri naturali a e b, analizzando in ordine decrescente i valori a partire dal più piccolo tra a e b (memorizzato nella variabile `minimo`), fino a trovarne uno che sia divisore di entrambi.

I valori da analizzare vengono calcolati e memorizzati nella variabile `d` per mezzo di un'istruzione iterativa. Non appena si trova un divisore comune si stampa un messaggio opportuno, e si assegna il valore 0 alla variabile `d` per arrestare l'iterazione; se invece il valore associato a `d` **non** è un divisore comune, `d` viene decrementata per fare in modo che nella successiva iterazione venga esaminato il valore immediatamente inferiore.

Esempi di programmi Python

```
a = eval(input("Primo numero: "))
b = eval(input("Secondo numero: "))
if a < b:
    minimo = a
else:
    minimo = b
d = minimo
while d > 0:
    if a % d == 0 and b % d == 0:
        print("Il MCD è", d)
        d = 0
    else:
        d = d - 1
```

Esempi di programmi Python

Una variante più sintetica (e più elegante) dello stesso programma (disponibile nel *file* `27_MCD_2.py`): l'istruzione iterativa ha il solo scopo di decrementare il valore associato a `d` finché questo **non** è un divisore comune di *a* e *b*. Notare che in questa versione il valore del MCD viene stampato **dopo** l'istruzione iterativa. Si può inoltre fare a meno della variabile `minimo`.

```
a = eval(input("Primo numero: "))
b = eval(input("Secondo numero: "))
if a < b:
    d = a
else:
    d = b
while not (a % d == 0 and b % d == 0):
    d = d - 1
print("Il MCD è", d)
```

Esempi di programmi Python

Un'altra variante del primo programma per il calcolo del MCD (disponibile nel *file* `28_MCD_3.py`) è mostrata di seguito: si usa una variabile (di nome `trovato`) per tener traccia del fatto che il MCD sia stato trovato o meno; nel primo caso la variabile dovrà assumere il valore `True`, nel secondo il valore `False`.

Il valore che la variabile `trovato` deve assumere **prima** dell'istruzione iterativa è `False`, poiché il MCD non è ancora stato trovato.

L'iterazione deve proseguire finché alla variabile `d` è associato un valore positivo **e** (`and`) il valore associato alla variabile `trovato` è `False`.

Notare che in questa versione non è necessario modificare in modo artificioso il valore associato a `d` per concludere l'iterazione quando si trova il MCD. Il risultato (il valore associato a `d`) può quindi essere stampato **al di fuori** dell'istruzione iterativa (cioè dopo che questa è terminata). Anche in questa versione si fa a meno della variabile `minimo`.

Esempi di programmi Python

```
a = eval(input("Primo numero: "))
b = eval(input("Secondo numero: "))
if a < b:
    d = a
else:
    d = b
trovato = False
while d > 0 and trovato == False:
    if a % d == 0 and b % d == 0:
        trovato = True
    else:
        d = d - 1
print("Il MCD è", d)
```

Esempi di programmi Python

Una quarta versione dello stesso programma (disponibile nel *file* `29_MCD_4.py`), nella quale si usa l'algoritmo di Euclide per il calcolo del MCD. In ogni passo dell'iterazione i valori delle variabili `a` e `b` vengono sostituiti dal valore più piccolo e dalla differenza tra il più piccolo e il più grande tra quelli dell'iterazione precedente, fino a che tali valori sono diversi.

```
a = eval(input("Primo numero: "))
b = eval(input("Secondo numero: "))
while a != b:
    if a < b:
        b = b - a
    else:
        a = a - b
print("Il MCD è", a)
```

Esempi di programmi Python

Il programma seguente (disponibile nel *file* `30_numeri_primi_1.py`) verifica se un dato numero naturale N sia primo o meno.

Contrariamente agli esempi precedenti, nei quali il risultato è un valore numerico (o una sequenza di cifre), in questo caso il risultato può essere visto come un **valore logico** (vero o falso), e può quindi essere rappresentato con i simboli `True` e `False`.

Il procedimento codificato da questo programma consiste nell'analisi (per mezzo di un'istruzione iterativa) di tutti i possibili divisori di N, cioè i valori compresi tra 2 e $\lfloor N/2 \rfloor$ (estremi inclusi, dove $\lfloor x \rfloor$ indica la parte intera di x), tenendo traccia, per mezzo della variabile `primo`, del fatto che sia già stato trovato o meno un divisore.

A tale variabile si assegna inizialmente il valore `True` (non è stato ancora trovato nessun divisore, quindi il numero viene considerato primo). Durante l'iterazione le si assegnerà il valore `False` se verrà trovato un divisore, e in questo caso l'iterazione potrà terminare senza completare l'analisi di tutti i possibili divisori. Il risultato viene mostrato per mezzo di un messaggio che dipende dal valore associato alla variabile `primo`.

Esempi di programmi Python

```
n = eval(input("Inserire un numero naturale: "))
primo = True
divisore = 2
while divisore <= n//2 and primo == True:
    if n % divisore == 0:
        primo = False
    else:
        divisore = divisore + 1
if n != 1 and primo == True:
    print("Il numero", n, "è primo")
else:
    print("Il numero", n, "non è primo")
```

Esempi di programmi Python

Il programma seguente stampa tutti i numeri primi compresi tra 1 e un dato numero naturale
(*file*: `31_sequenza_numeri_primi_1.py`)

A questo scopo sono necessarie due istruzioni iterative nidificate: la prima genera tutti i numeri dell'intervallo da considerare, la seconda (nidificata) verifica se il numero in esame sia primo (sfruttando il programma dell'esempio precedente).

Esempi di programmi Python

```
n = eval(input("Inserire un numero naturale: "))
print("I numeri primi tra 1 e", n, "sono:")
k = 1
while k <= n:
    primo = True
    divisore = 2
    while divisore <= k//2 and primo == True:
        if k % divisore == 0:
            primo = False
        else:
            divisore = divisore + 1
    if k != 1 and primo == True:
        print(k)
    k = k + 1
```

Esempi di programmi Python

Il programma di quest'ultimo esempio (disponibile nel *file* `32_polinomio.py`) calcola il valore di un polinomio di grado qualsiasi:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

per un dato valore della variabile x. I dati d'ingresso sono il grado del polinomio, il valore di x e i valori dei coefficienti.

Nella definizione del programma si è scelto di acquisire i coefficienti, per mezzo di un'istruzione iterativa, a partire da quello del termine di grado **inferiore**: in questo modo le varie potenze della variabile x possono essere facilmente calcolate nella stessa istruzione iterativa.

Esempi di programmi Python

```
n = eval(input("Grado del polinomio: "))
x = eval(input("Valore della variabile: "))
polinomio = 0
potenza_x = 1
k = 0
while k <= n:
    print("Inserire il coefficiente di grado", k)
    a = eval(input())
    polinomio = polinomio + a*potenza_x
    potenza_x = potenza_x*x
    k = k + 1
print("Il valore del polinomio è", polinomio)
```

Esercizi

Scrivere programmi Python che realizzino le seguenti operazioni e stampino i risultati nella *shell*

- calcolare il valore più grande in una sequenza di numeri qualsiasi (dati d'ingresso: la lunghezza della sequenza e i suoi elementi)
- calcolare le cifre della rappresentazione di un numero naturale in una base qualsiasi (dati d'ingresso: il numero da rappresentare e la base di numerazione)
- stampare tutti i divisori di un dato numero naturale
- calcolare il minimo comune multiplo (MCM) di due numeri naturali (dati d'ingresso: i valori dei due numeri) (suggerimento: il MCM tra due numeri a e b è definito come il più piccolo numero naturale del quale sia a che b siano divisori)

L'istruzione `break`

Si è visto che l'esecuzione di un'istruzione iterativa termina quando, all'inizio di un'iterazione, l'espressione condizionale risulta falsa.

Per concludere l'esecuzione di un'istruzione iterativa è anche possibile usare l'istruzione `break`. Questa istruzione può essere usata **solo** all'interno di un'istruzione iterativa, in un qualsiasi punto della sequenza di istruzioni da ripetere.

Di norma l'istruzione `break` viene scritta all'interno di un'istruzione condizionale nidificata, per concludere l'esecuzione dell'istruzione iterativa nel caso in cui si verifichi una data condizione.

Quando l'interprete incontra l'istruzione `break`, esso conclude immediatamente l'esecuzione dell'istruzione iterativa e passa a eseguire l'eventuale istruzione successiva.

L'istruzione `break`: esempio

Un chiaro esempio dell'utilità dell'istruzione break è l'algoritmo visto in precedenza per determinare se un numero naturale sia primo (si veda il *file* `30_numeri_primi_1.py`).

Nella versione che segue (*file*: `33_numeri_primi_2.py`) l'iterazione viene conclusa con `break` non appena viene trovato un divisore. In questo modo non è necessario aggiungere la condizione `primo == True` nell'espressione condizionale dell'istruzione `while`. Inoltre l'istruzione `divisore = divisore + 1` può essere scritta **al di fuori** dell'istruzione condizionale, poiché l'istruzione iterativa può proseguire solo se **non** è ancora stato trovato un divisore.

L'istruzione break: esempio

```
n = eval(input("Inserire un numero naturale: "))
primo = True
divisore = 2
while divisore <= n//2:
    if n % divisore == 0:
        primo = False
        break
    divisore = divisore + 1
if n != 1 and primo == True:
    print("Il numero", n, "è primo")
else:
    print("Il numero", n, "non è primo")
```

L'istruzione `break`: esercizi

Riscrivere i programmi per il calcolo del MCD e del MCM di due numeri naturali, nella versione basata sull'analisi di tutte le possibili soluzioni, usando l'istruzione `break`.

Funzioni

Le *funzioni* nei linguaggi di alto livello

Le funzioni direttamente disponibili in un linguaggio di programmazione come Python (per esempio, `print` e `input`) sono dette **predefinite**, o ***built-in***.

Molte altre funzioni, che costituiscono la cosiddetta **libreria** di un linguaggio, sono rese disponibili per operazioni di utilità generale, come per esempio, nel caso di Python

- operazioni matematiche come il calcolo di grandezze trigonometriche e logaritmi
- operazioni sulle stringhe, come il calcolo della lunghezza

Il termine "funzione" è stato introdotto per analogia con il concetto di funzione matematica, poiché spesso le funzioni dei linguaggi di programmazioni elaborano un determinato insieme di valori di ingresso e producono un valore (per es., un numero o una stringa) come risultato.

Le *funzioni* nei linguaggi di alto livello

Altre funzioni di libreria consentono di eseguire operazioni quali

- la realizzazione di interfacce grafiche per i propri programmi
- la generazione di numeri casuali

I linguaggi di programmazione consentono inoltre ai programmatori di definire **nuove** funzioni da usare nei propri programmi, dette funzioni **definite dall'utente** (***user-defined***).

Utilità delle funzioni

La disponibilità di funzioni **predefinite** o **di libreria** evita agli utenti di dover scrivere propri programmi per realizzare operazioni di utilità generale.

Inoltre, la possibilità di definire **nuove** funzioni presenta diversi vantaggi

- consente di semplificare la scrittura di programmi complessi suddividendoli in più parti (funzioni), ciascuna delle quali svolge un compito **distinto** dalle altre, e può essere sviluppata in modo **indipendente** da esse
- l'esecuzione di una funzione può essere richiesta in diversi punti di uno stesso programma: ciò evita di scrivere più volte le istruzioni corrispondenti
- una **stessa** funzione può essere usata in programmi diversi

La libreria del linguaggio Python

Il linguaggio Python comprende un vasto insieme di funzioni predefinite e di libreria.

La descrizione completa si trova nella sezione *Library reference* del sito web contenente la documentazione ufficiale del linguaggio (in inglese): `https://docs.python.org/3/`.

Le funzioni **predefinite** (come `print` e `input`) sono direttamente accessibili dalla *shell* e dai propri programmi.

Le funzioni **di libreria** sono suddivise in diversi `moduli`, e per poterle usare è necessaria una specifica istruzione, `import`, che verrà descritta più avanti.

Caratteristiche delle funzioni

A ogni funzione è associato un **nome simbolico** (analogo ai nomi delle variabili), detto **nome della funzione**.

Ogni funzione può elaborare un determinato numero di valori di ingresso, detti **argomenti** per analogia con gli argomenti delle funzioni matematiche.

Ogni funzione può infine **restituire** un valore di un determinato tipo, come **risultato** dell'elaborazione svolta sugli argomenti.

Come casi particolari, una funzione può non ricevere nessun argomento o non restituire nessun valore.

Esecuzione di una funzione: *chiamata*

L'esecuzione di una funzione si ottiene attraverso una specifica **espressione**, detta **chiamata** (o **invocazione**) **di funzione**.

Sintassi: nome-funzione(arg_1 , arg_2 , . . . , arg_n)

- arg_1, . . . , arg_n sono **espressioni**, i cui **valori** costituiranno gli argomenti della funzione
- il **numero** degli argomenti dipende dalla specifica funzione: se il numero degli argomenti presenti nella chiamata non corrisponde a quello previsto si otterrà un messaggio d'errore
- anche il **tipo** di ciascun argomento deve coincidere con quello previsto dalla funzione
- di norma, la chiamata di una funzione produce, o **restituisce**, un **valore** (come ogni espressione)

Principali funzioni Python predefinite: `input`

La funzione `input` è stata descritta in precedenza.

Essa può non ricevere argomenti, oppure può riceverne uno (di norma una stringa, anche se può trattarsi di un'espressione di tipo qualsiasi)

- `input()`
- `input(arg)`

La chiamata di `input` produce la stampa nella *shell* del valore associato ad arg (se presente), l'acquisizione della sequenza di caratteri che verranno inseriti nella stessa *shell* attraverso la tastiera, fino alla pressione del tasto `INVIO`, e la restituzione di tale sequenza sotto forma di una **stringa**.

Principali funzioni Python predefinite: `print`

Come si è già visto, la funzione `print` consente di stampare nella *shell* dell'ambiente di programmazione (o del sistema operativo) i valori di una o più espressioni Python

- `print(`espressione`)`
- `print(`espressione$_1$`,` espressione$_2$`,` `...` `)`

Nel secondo caso i valori delle espressioni vengono stampati su una **stessa** riga, separati da un carattere di spaziatura.

Si noti che `print` è una funzione particolare, in quanto non **restituisce** nessun valore: il suo unico scopo è **stampare** valori nella *shell*.

Principali funzioni Python predefinite

Altre funzioni predefinite di utilità generale sono le seguenti

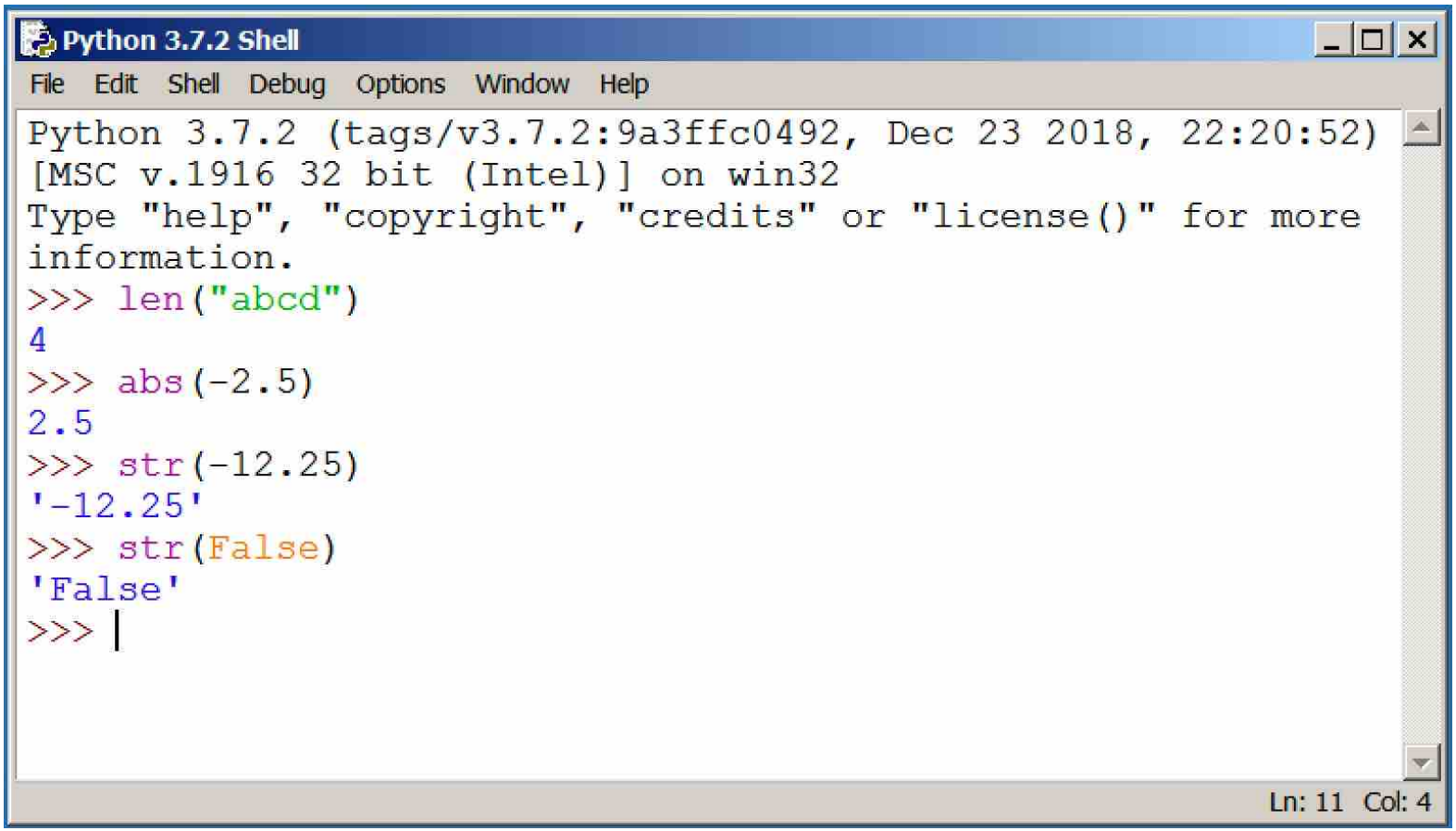
- `abs(`numero`)`
  restituisce il valore assoluto di un numero
- `len(`stringa`)`
  restituisce il numero di caratteri di una stringa
- `str(`espressione`)`
  restituisce una stringa composta dalla sequenza di caratteri corrispondenti alla rappresentazione del valore di espressione (che può essere di un qualsiasi tipo: numero, stringa, valore logico, ecc.)
- `eval(`stringa`)`
  se stringa contiene una qualsiasi espressione **valida** del linguaggio Python, restituisce il valore di tale espressione

(cont.)

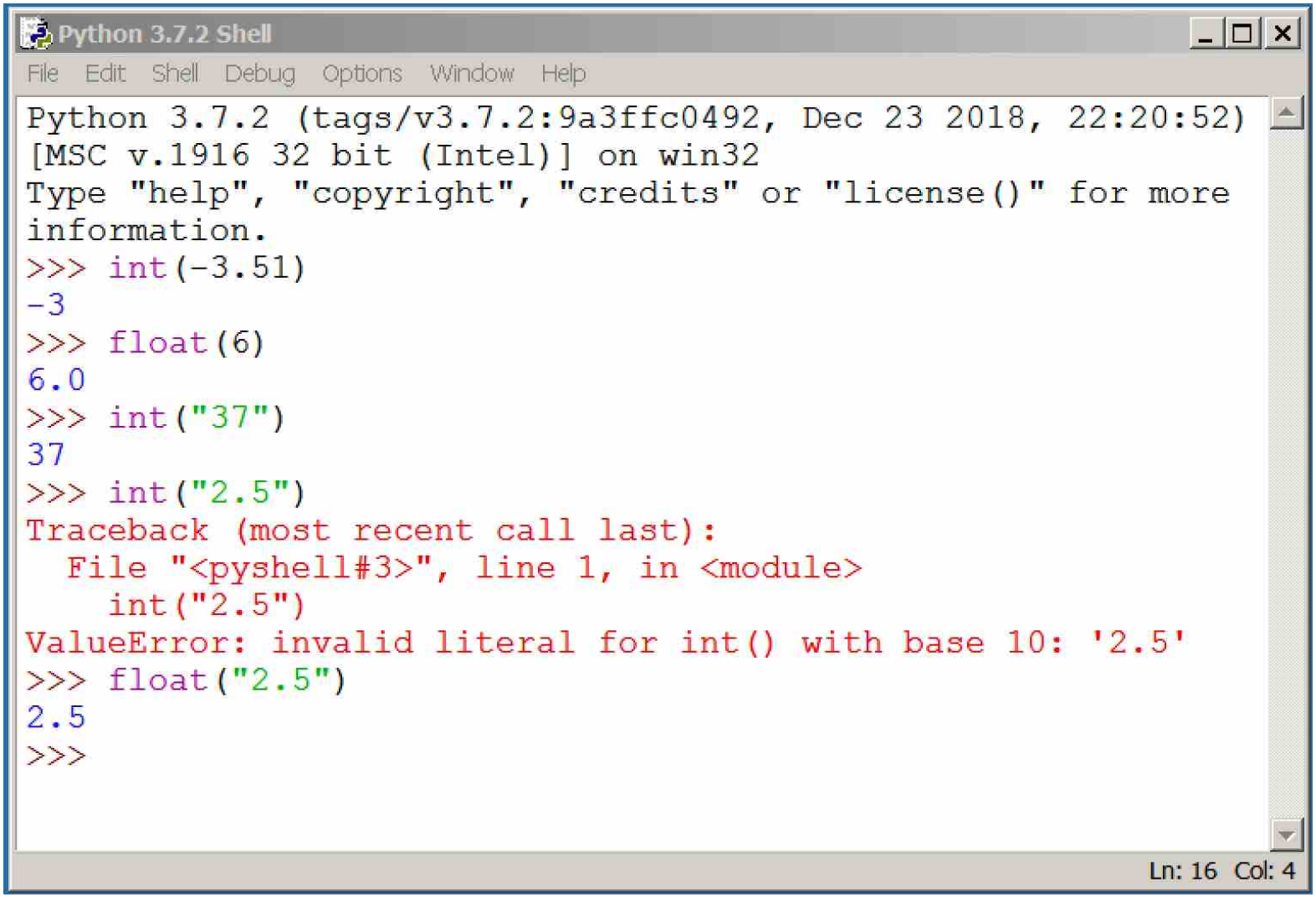
Principali funzioni Python predefinite

- `int(numero)`
  restituisce la parte intera di un numero
- `float(numero)`
  restituisce il valore di numero come numero frazionario (*floating point*)
- `int(stringa)`
  se stringa contiene la rappresentazione di un numero **intero**, restituisce il numero corrispondente a tale valore
- `float(stringa)`
  se stringa contiene la rappresentazione di un numero qualsiasi (sia intero che frazionario), restituisce il suo valore espresso come numero frazionario

Principali funzioni Python predefinite: esempi



Principali funzioni Python predefinite: esempi

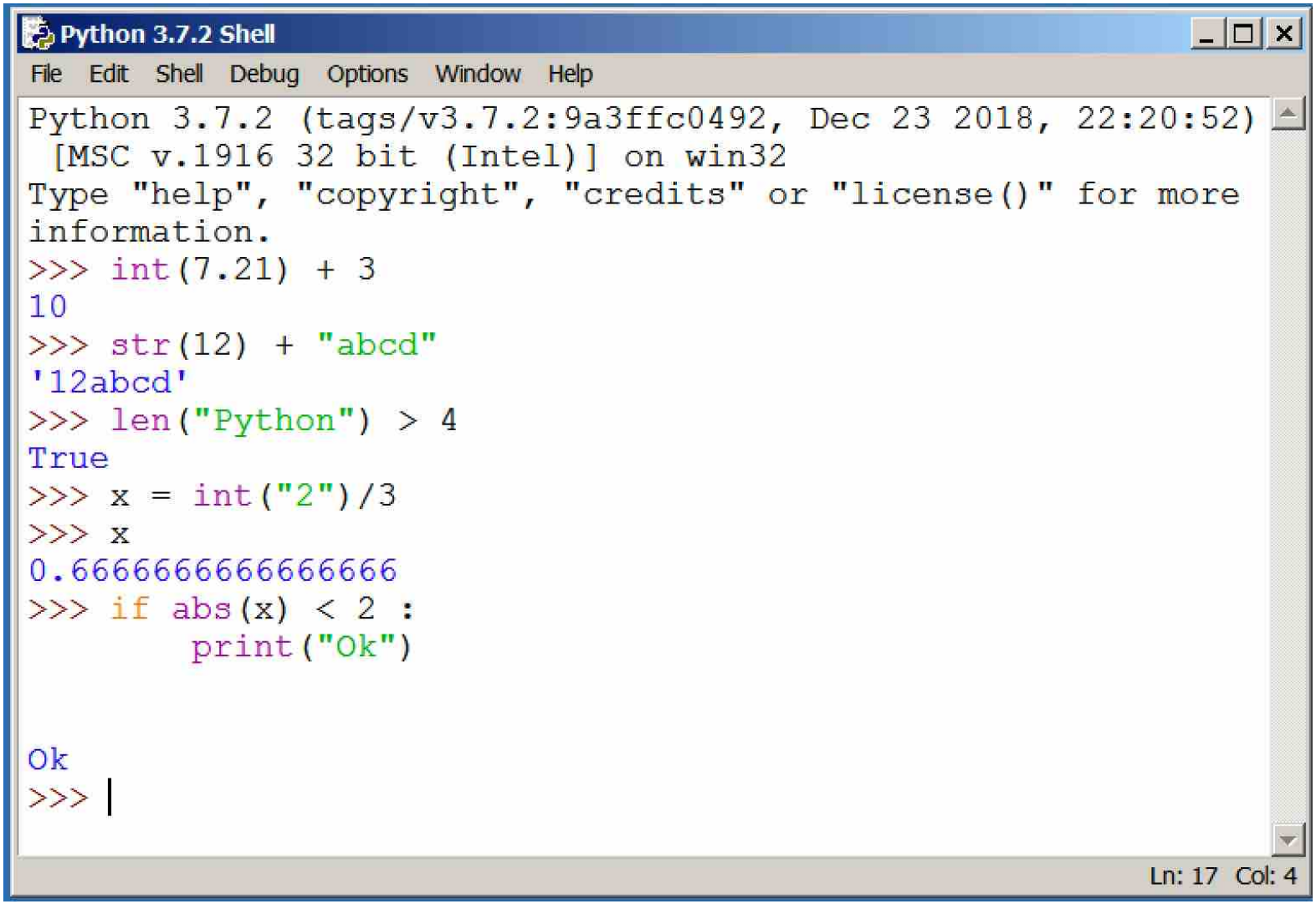


Ancora sulla sintassi della chiamata di funzione

Dal punto di vista sintattico la chiamata di una funzione è un'**espressione**, e deve quindi rispettare le stesse regole sintattiche viste in precedenza per le espressioni

- può essere scritta direttamente nella *shell* (come negli esempi precedenti): in questo caso anche il valore restituito dalla funzione verrà mostrato nella *shell*
- può comparire come **operando** di una qualsiasi espressione più complessa (aritmetica, logica, o composta da stringhe)
- può comparire nell'espressione di un'istruzione di **assegnamento**
- può comparire nell'**espressione condizionale** all'interno di un'istruzione condizionale o iterativa

Chiamata di funzione: esempi

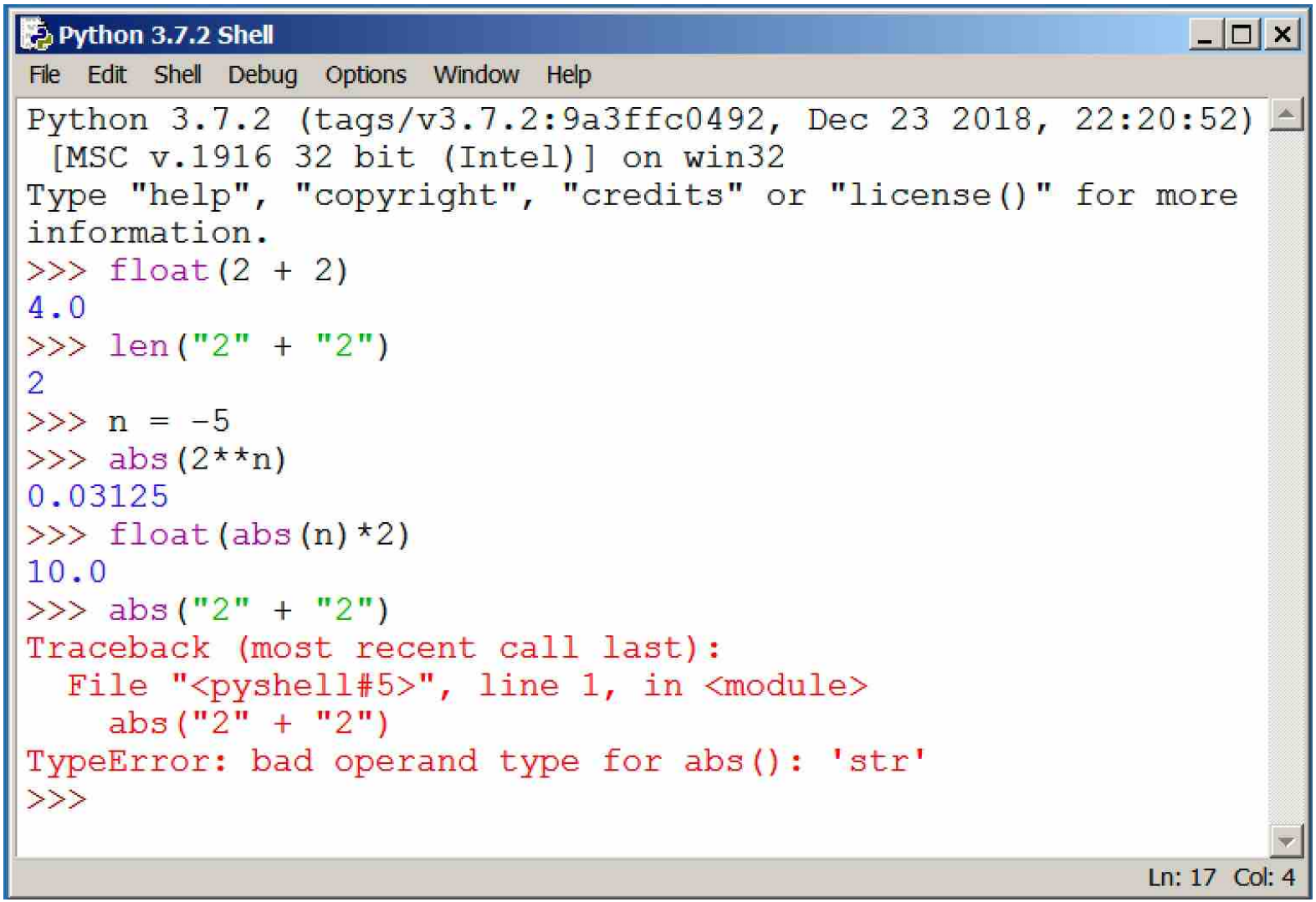


Chiamata di funzione: sintassi degli argomenti

Gli argomenti di una chiamata di funzione sono a loro volta **espressioni** qualsiasi, il cui valore deve essere del tipo previsto dalla specifica funzione (in caso contrario si otterrebbe un messaggio d'errore).

Ne consegue come caso particolare che una chiamata di funzione può contenere come argomenti altre chiamate di funzione: in tal caso si parla di chiamate **nidificate**.

Argomenti delle chiamate di funzione: esempi



La libreria di funzioni del linguaggio Python

Le funzioni di libreria di Python sono suddivise in diversi **moduli**, ciascuno dei quali è identificato univocamente da un **nome simbolico**.

Per esempio, le funzioni che eseguono operazioni matematiche fanno parte di un modulo di nome `math`, mentre il modulo `random` comprende funzioni per la generazione di numeri casuali.

Principali funzioni della libreria `math`

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

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

Principali funzioni della libreria `random`

funzione	descrizione
`random()`	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 generato)
`uniform(`a`,`b`)`	come sopra, nell'intervallo $[a, b)$ (gli argomenti sono numeri qualsiasi)
`randint(`a`,`b`)`	genera un numero **intero** nell'insieme $\{a, \ldots, b\}$ da una distribuzione di probabilità **uniforme** (gli argomenti devono essere numeri **interi**)

Ogni chiamata delle tre funzioni sopra descritte produce un numero **pseudo-casuale**, ovvero ottenuto mediante un algoritmo che genera sequenze di numeri che "appaiono" casuali.

L'istruzione `import`

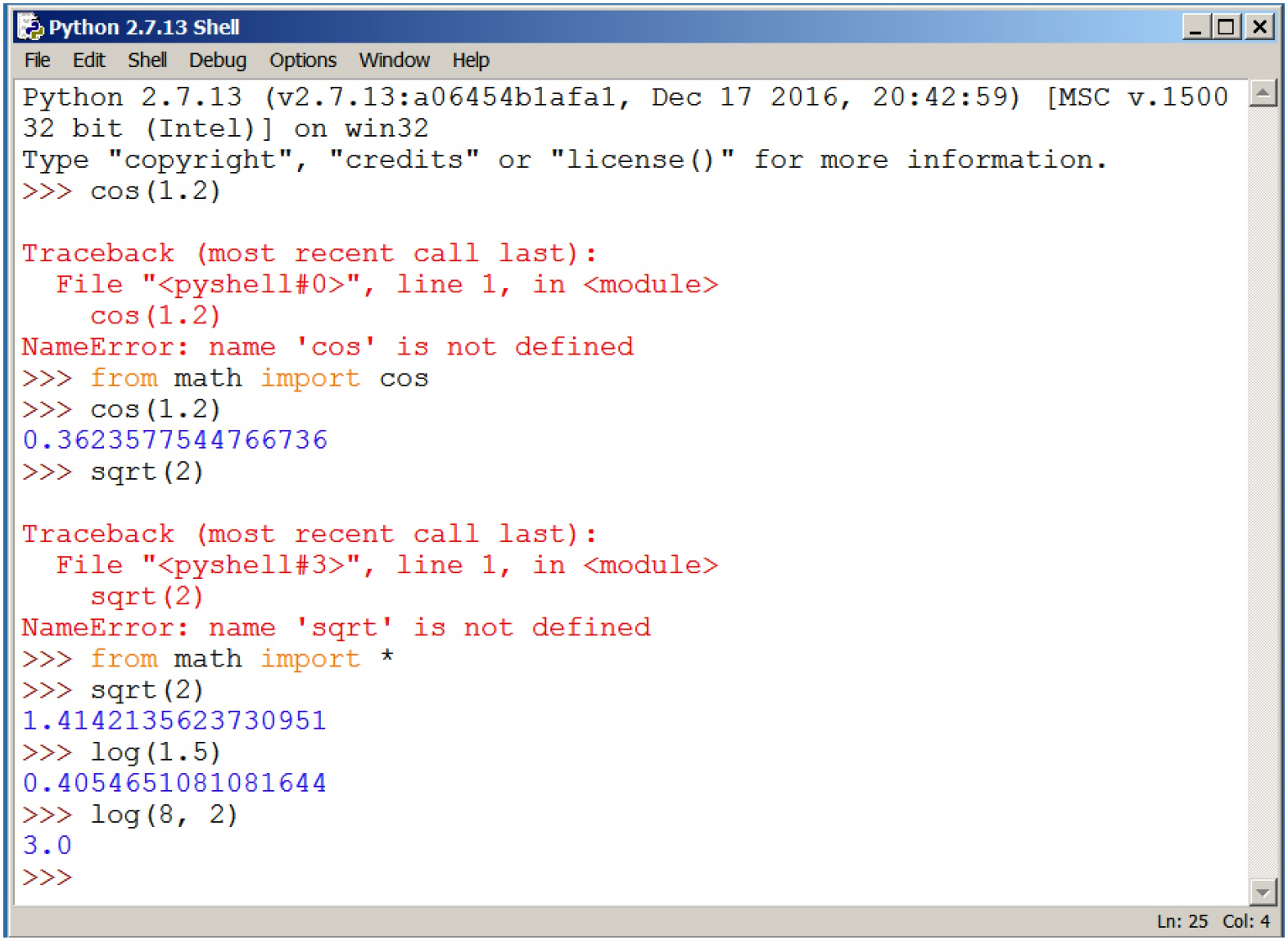
La chiamata di una funzione appartenente a un modulo come `math` e `random` (sia nella *shell* che in un programma) può essere eseguita solo se **prima** è stata "comunicata" all'interprete l'intenzione di usare funzioni di tali moduli. Ciò avviene mediante l'istruzione `import`, che prevede la seguente **sintassi**:

`from` nome-modulo `import` nome-funzione

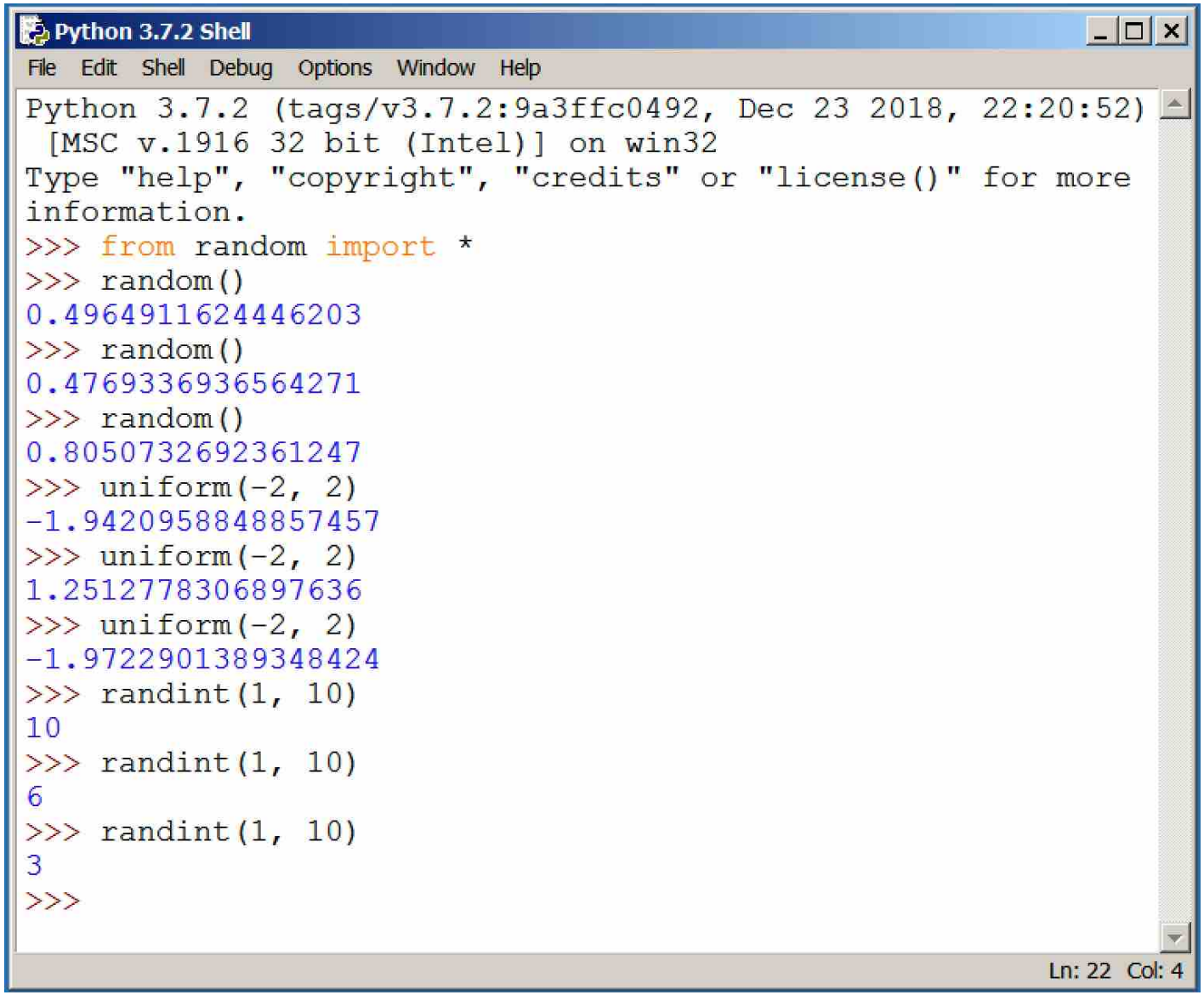
- nome-modulo è il nome simbolico di un modulo
- nome-funzione può essere
  - il nome di una specifica funzione di tale modulo, o i nomi di più funzioni (separati da virgole): questo consentirà di usare solo la funzione o le funzioni specificate
  - il simbolo `*` indicante **tutte** le funzioni di tale modulo

Senza tale istruzione, ogni chiamata produrrà un errore, come mostrato negli esempi seguenti.

L'istruzione `import`: esempi



L'istruzione `import`: esempi



Il modulo `math`: costanti matematiche notevoli

Nel modulo `math` sono anche definite due **variabili** alle quali è associato il valore (approssimato) delle costanti matematiche π ($\approx 3{,}14$) ed e (la base dei logaritmi naturali, $\approx 2{,}71$):

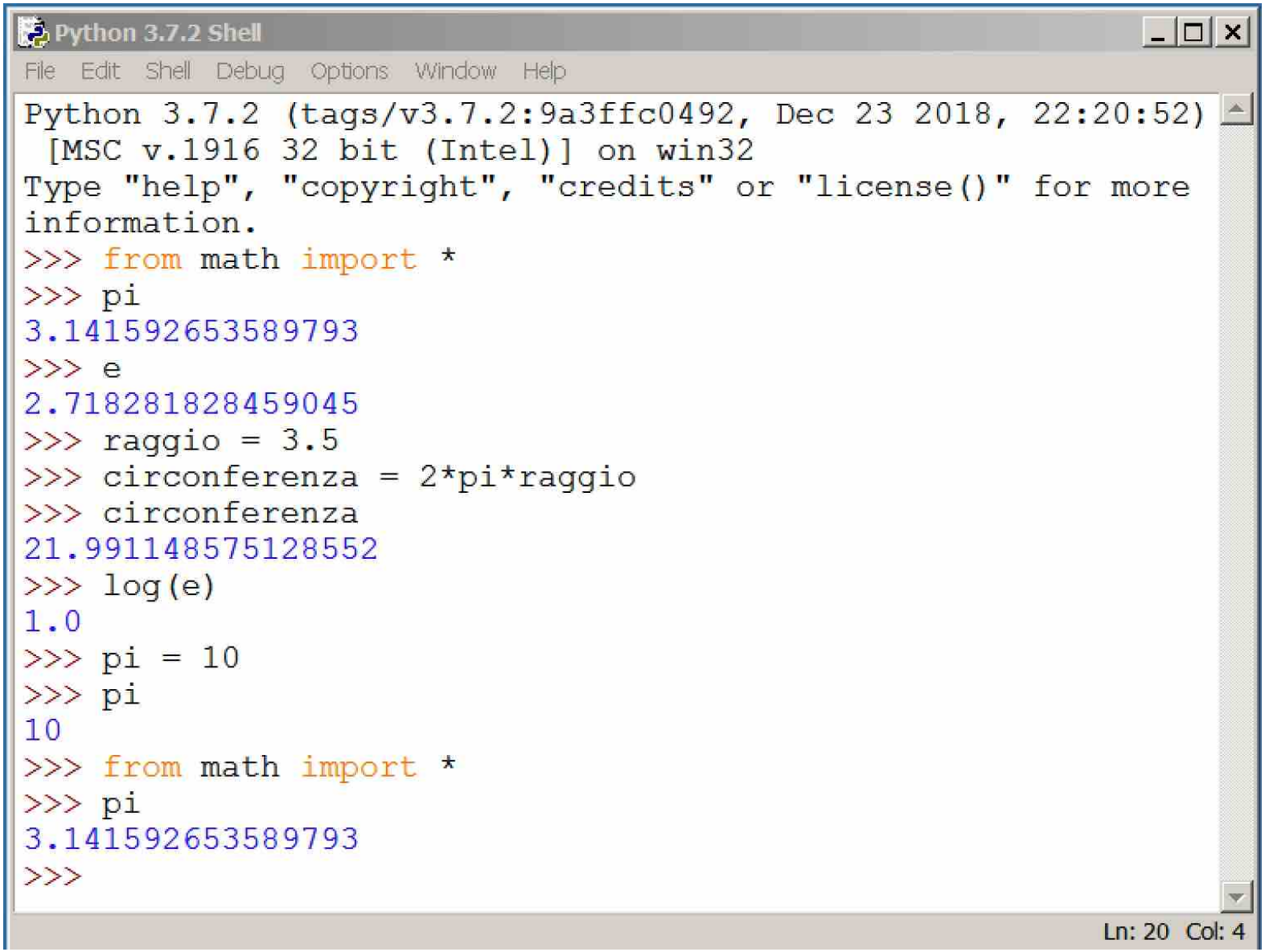
- `pi`
- `e`

Anche per usare queste variabili è necessaria l'istruzione `import`, per esempio

- `from math import *`
- `from math import pi, e`

Nota: Il valore associato alle due variabili può essere modificato con istruzioni di assegnamento, anche se ciò non è consigliabile. Il loro valore originale può essere ripristinato eseguendo di nuovo l'istruzione `import`.

Le variabili pi ed e: esempi



Definizione di nuove funzioni

Nel linguaggio Python 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'istruzione `def`, che può essere scritta sia nella *shell* che in un programma (in una finestra dell'*editor*).

Definizione di nuove funzioni: l'istruzione `def`

Sintassi:

`def` nome-funzione`(`par_1`, ..., `par_n`):`
    corpo della funzione

- nome-funzione è un nome simbolico scelto dal programmatore, con gli stessi vincoli a cui sono soggetti i nomi delle variabili
- par_1, ..., par_n 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.

Definizione di nuove funzioni: l'istruzione `return`

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

Sintassi: `return` espressione
dove espressione è un'espressione Python **qualsiasi**.

Questa istruzione può essere usata **solo** all'interno di una funzione.

Se una funzione **non** deve restituire nessun valore

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

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

Il **riavvio** della *shell* a seguito dell'esecuzione di un programma comporta la "cancellazione" delle eventuali funzioni, oltre che delle variabili, definite in precedenza.

Chiamata di una funzione: meccanismo di esecuzione

Anche per le funzioni definite dall'utente il numero di argomenti indicati nella chiamata dovrà corrispondere al numero di argomenti previsti nella definizione, cioè al numero dei parametri indicati nell'intestazione.

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

- **associa** il **valore** di ciascun **argomento** al **parametro** corrispondente (quindi a tali variabili **è già associato** un valore nel momento in cui inizia l'esecuzione della funzione)
- esegue le istruzioni del corpo della funzione, fino a incontrare l'istruzione `return`, oppure l'ultima istruzione del corpo
- se l'eventuale istruzione `return` è seguita da un'espressione, **restituisce** il valore di tale espressione come **risultato** della chiamata

Definizione di funzioni: esempio

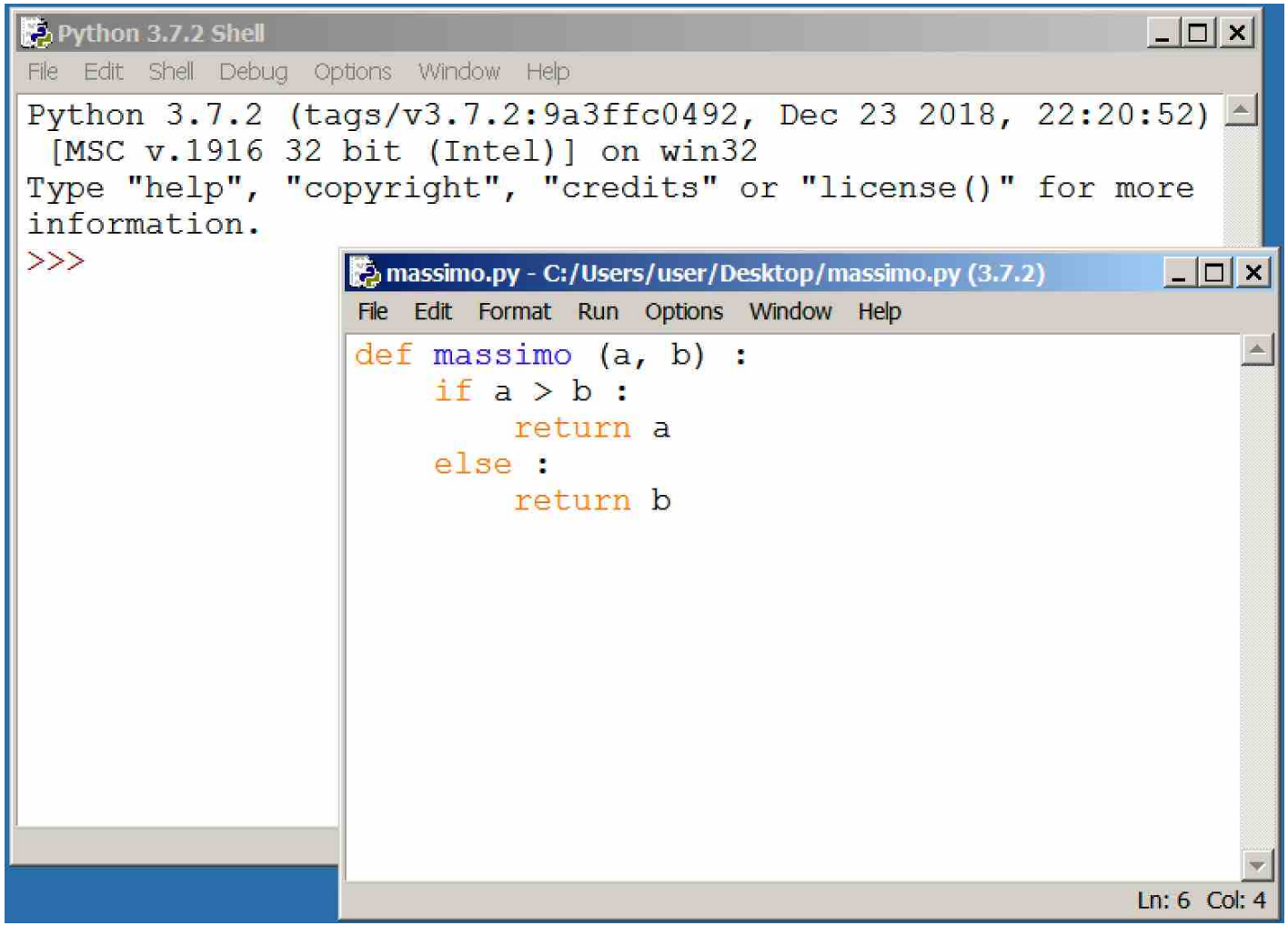
Si supponga di voler definire una funzione che restituisca il più grande tra due numeri ricevuti come argomenti.

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
```

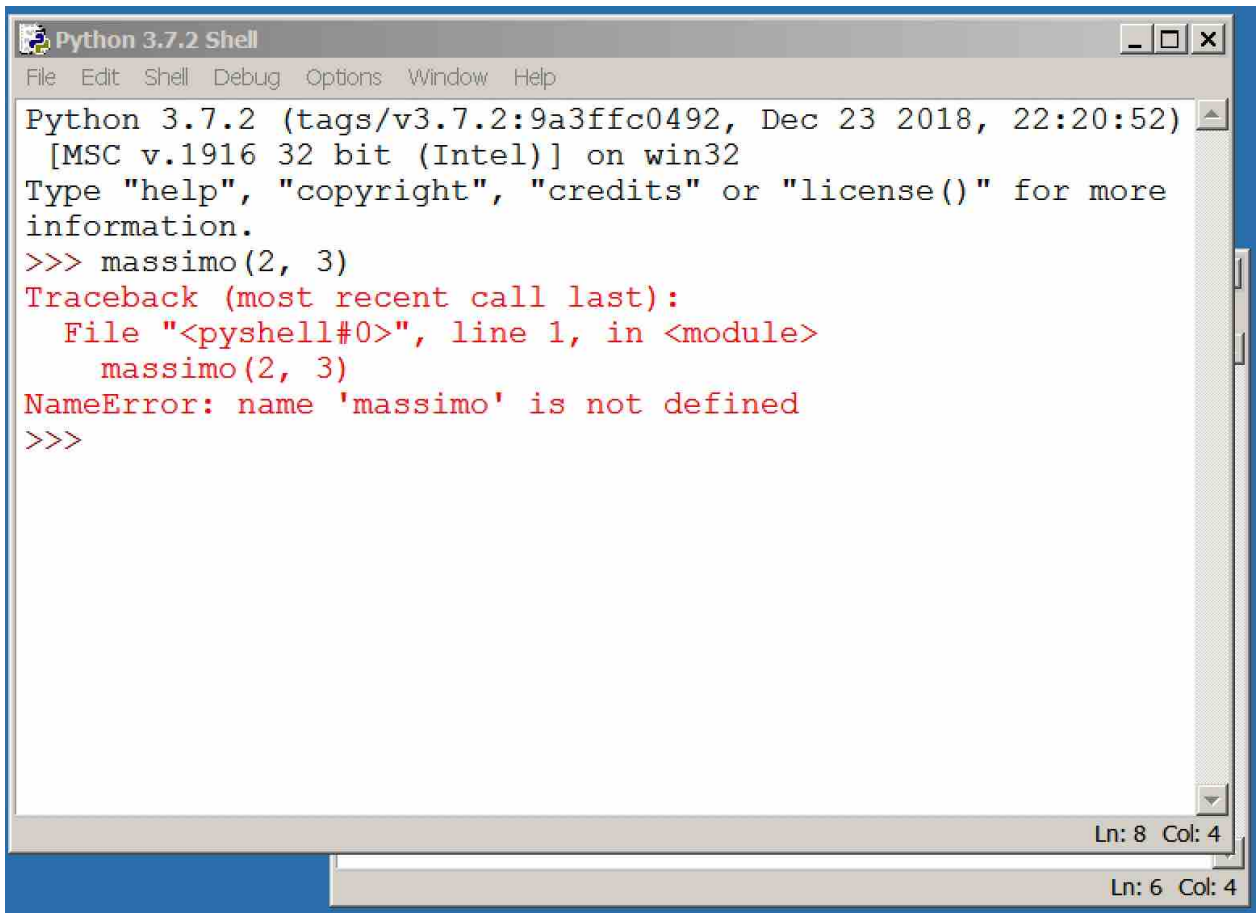
Definizione di funzioni: esempio

La definizione di una funzione deve essere scritta in una finestra dell'*editor*, e deve essere salvata in un *file*



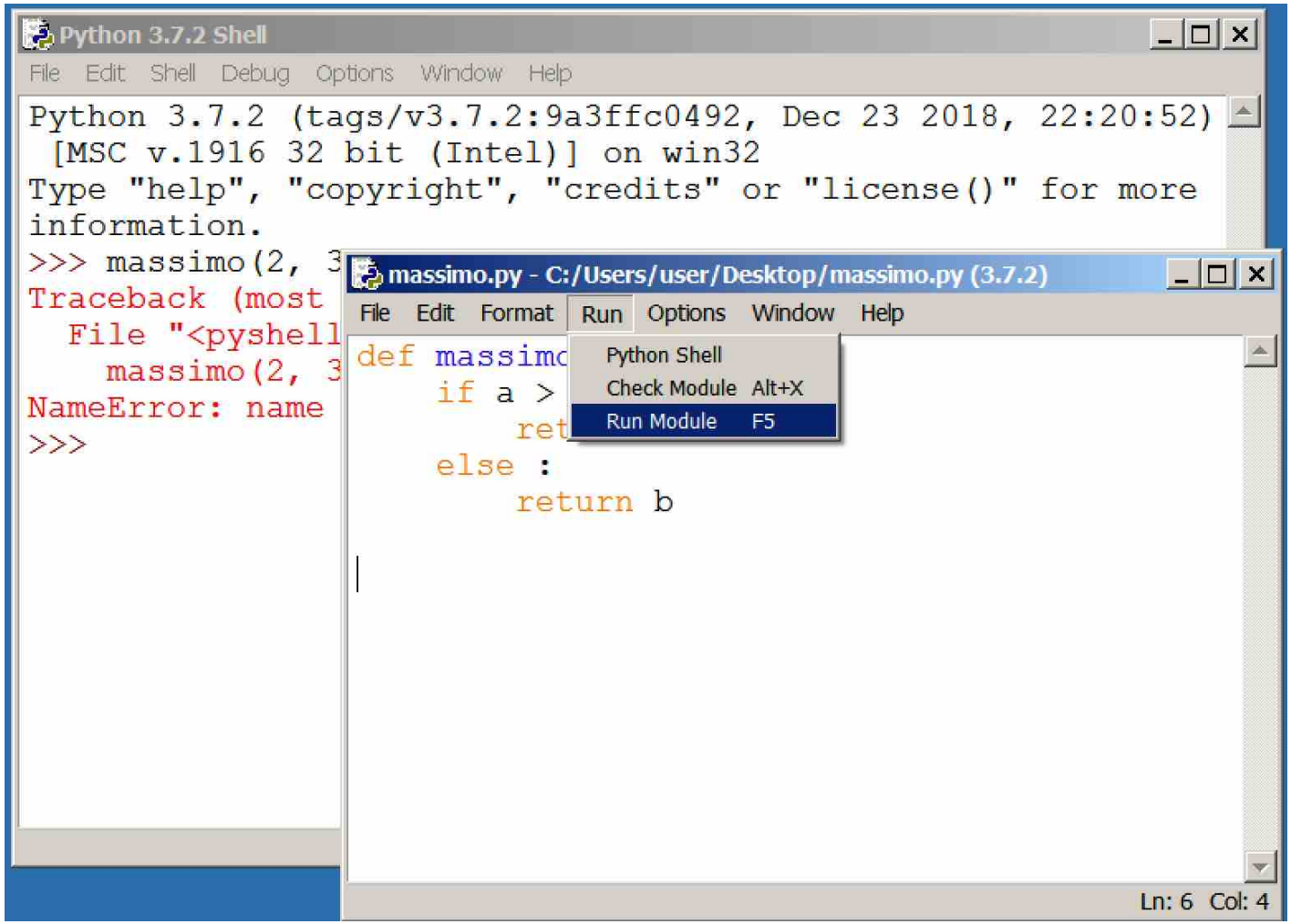
Definizione di funzioni: esempio

Dopo il salvataggio del *file*, la funzione che esso contiene non è ancora "nota" all'interprete: una sua chiamata scritta nella *shell* produrrà un messaggio d'errore



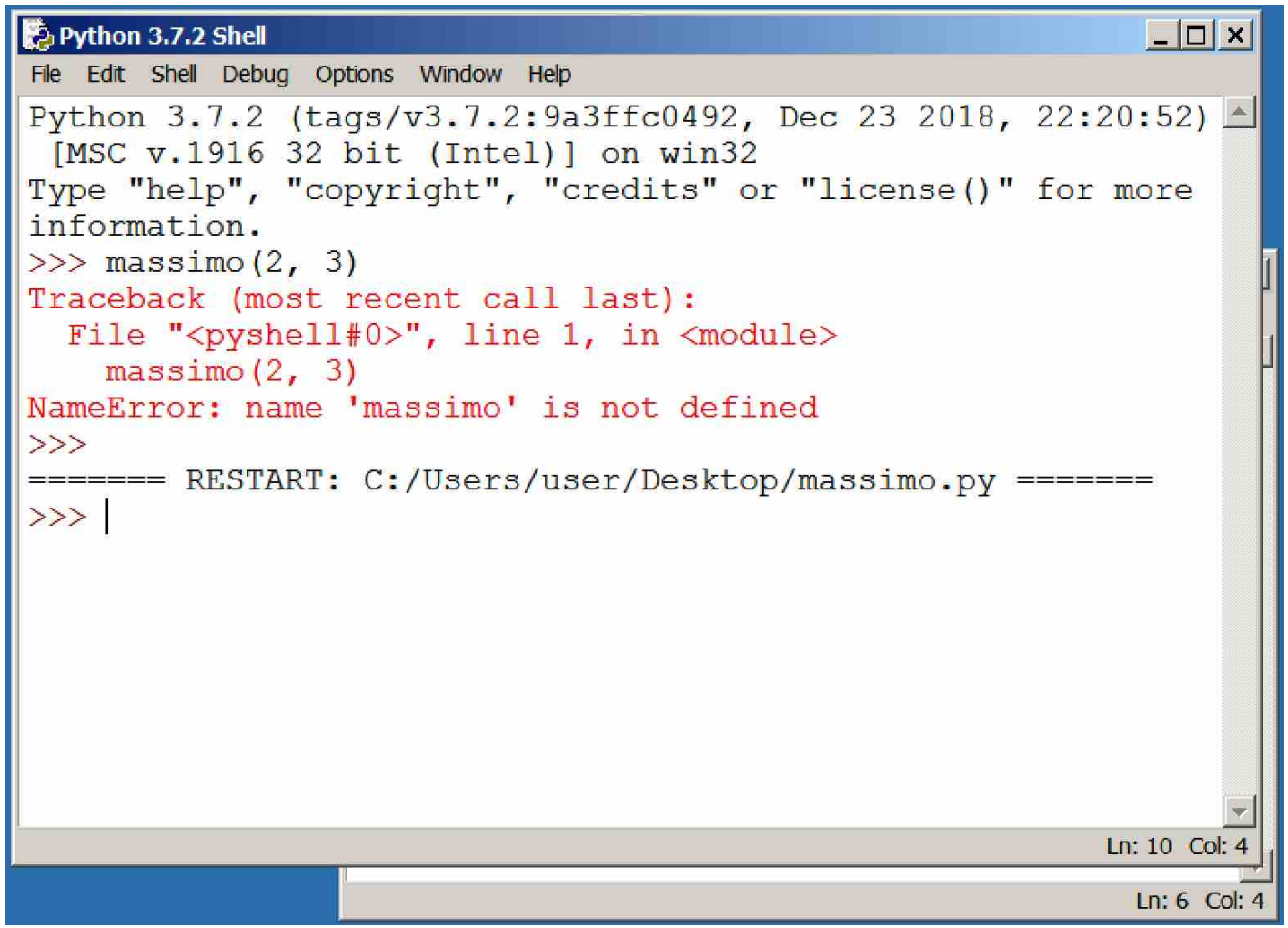
Definizione di funzioni: esempio

Prima di chiamare una funzione è necessario eseguire l'istruzione `def` nel *file* che ne contiene la definizione



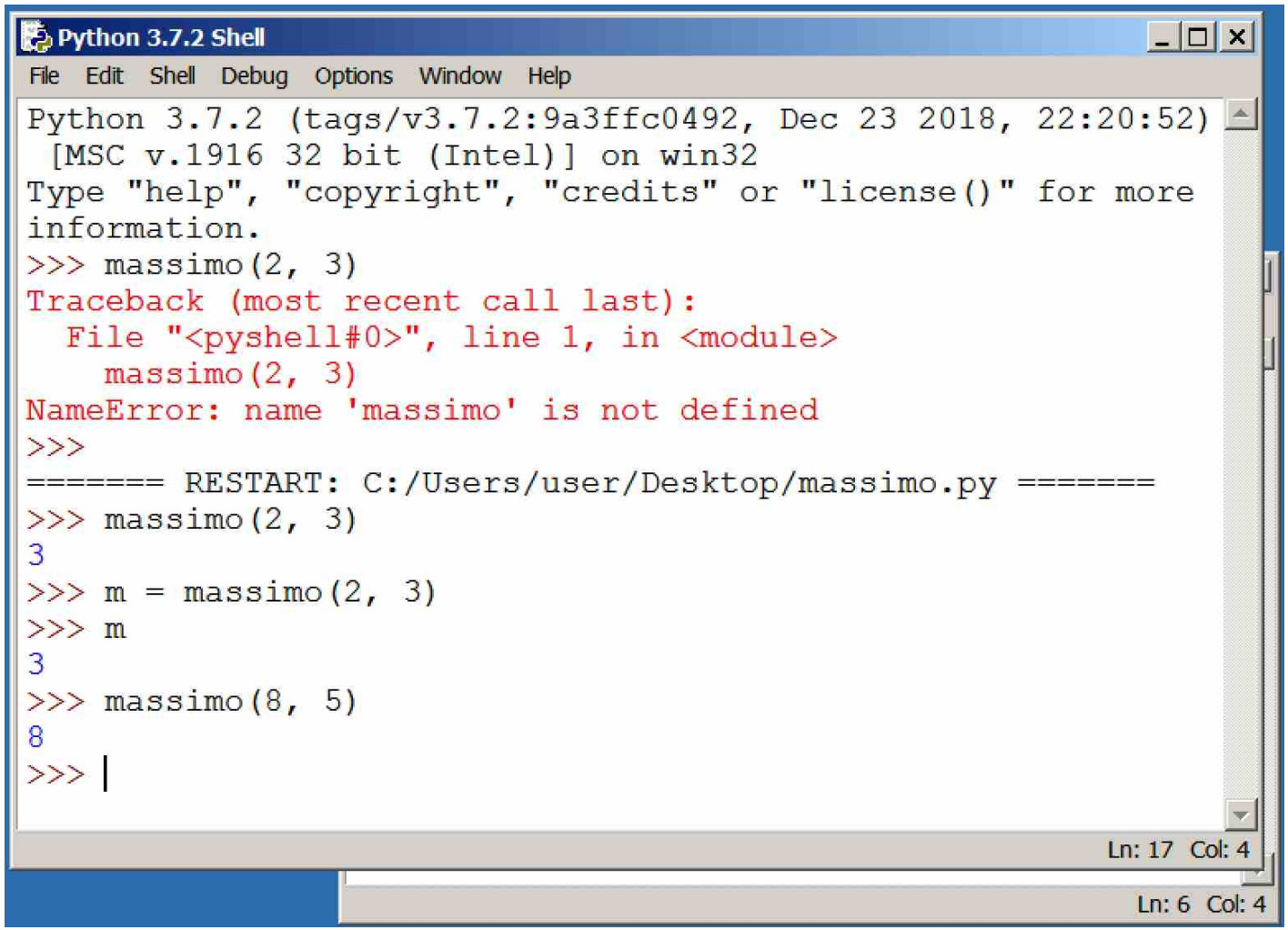
Definizione di funzioni: esempio

Si ricordi che l'esecuzione dell'istruzione `def` non comporta l'esecuzione delle istruzioni della funzione



Definizione di funzioni: esempio

Da questo momento è possibile chiamare la funzione dalla *shell*



Ancora sui parametri di una funzione

Come si è detto in precedenza, quando una funzione viene chiamata, prima di iniziare l'esecuzione delle sue istruzioni l'interprete assegna ai parametri i valori degli argomenti indicati nella chiamata.

Nell'esempio precedente, alle variabili (parametri) `a` e `b` della funzione `massimo` **sarà già associato un valore** nel momento in cui inizierà l'esecuzione delle istruzioni del corpo della funzione.

Questo significa che i valori delle variabili che costituiscono i parametri della funzione **non** devono essere assegnati per mezzo di istruzioni di assegnamento nel **corpo** della funzione, ma vengono indicati nella **chiamata** della stessa funzione.

Definizione di funzioni: esempi

In precedenza si sono mostrati esempi di programmi per il calcolo del massimo comun divisore tra due numeri naturali con l'algoritmo di Euclide, e della somma dei primi m termini della serie armonica (per un dato valore di m).

Di seguito si mostra come gli stessi programmi possono essere scritti sotto forma di funzioni. In entrambi i casi

- i **valori d'ingresso**, che i programmi acquisivano mediante `input`, sono definiti come **argomenti**
- il **risultato**, che i programmi stampavano nella *shell* mediante `print`, viene **restituito** mediante l'istruzione `return`

Le funzioni possono essere definite e poi chiamate come mostrato nell'esempio precedente.

Definizione di funzioni: esempi

Calcolo del massimo comun divisore con l'algoritmo di Euclide (disponibile nel *file* `34_MCD_funzione.py`)

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

Definizione di funzioni: esempi

Calcolo della somma dei primi *m* termini della serie armonica (disponibile nel *file* `35_serie_armonica_funzione.py`)

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

Funzioni senza argomenti o senza risultato

Come caso particolare è possibile definire funzioni che **non ricevono argomenti** (come nel caso della funzione predefinita `random` della libreria omonima, descritta in precedenza), oppure funzioni che **non restituiscono un risultato**.

Lo scopo di una funzione che non restituisce un risultato potrebbe essere, per esempio, quello di **stampare** un messaggio nella *shell*.

- nell'intestazione e nelle chiamate di una funzione che **non riceve argomenti** si dovranno scrivere le parentesi tonde senza il nome di alcun parametro o senza argomenti al loro interno
- nella corpo della definizione di una funzione che **non restituisce un risultato** non si dovrà usare nessuna istruzione del tipo:

  `return` espressione

Definizione di funzioni: esempio

Un semplice esempio di funzione che non riceve argomenti **e** non restituisce un risultato: il suo scopo è stampare il messaggio `Buongiorno` nella *shell*

```
def saluto():
    print("Buongiorno")
```

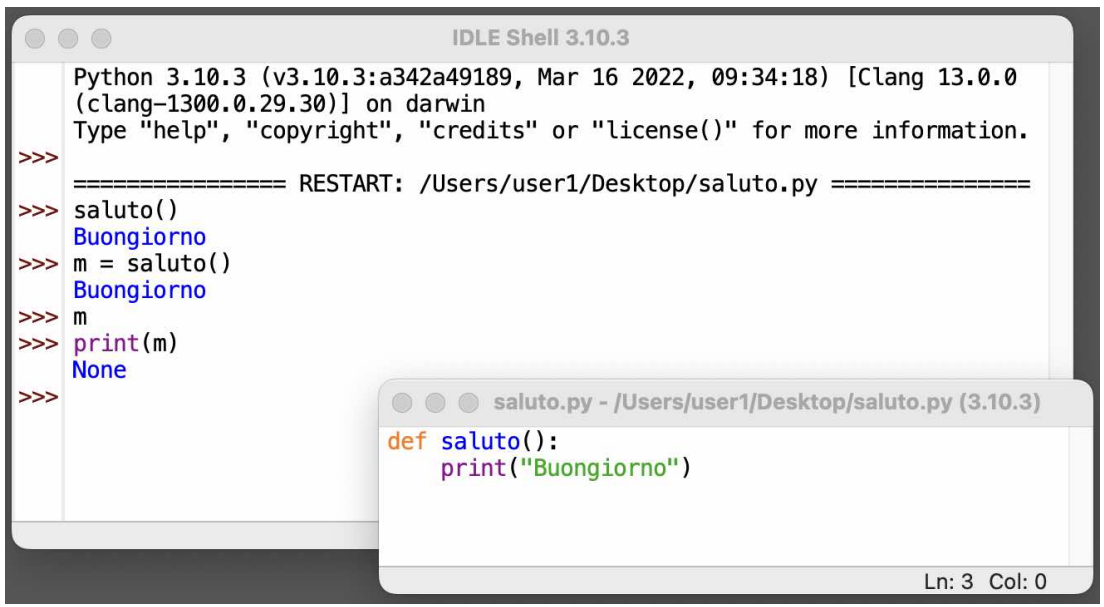
Il fatto che questa funzione non restituisca un risultato può essere evidenziato scrivendo nella *shell* una chiamata come la seguente (**dopo** aver eseguito l'istruzione `def` per definire la funzione):

```
m = stampa()
```

Si provi poi a mostrare nella *shell* il valore associato a `m`, oppure a stamparlo con `print(m)`, come mostrato di seguito.

Definizione di funzioni: esempio

Come si può vedere, l'espressione `m` non produce alcun risultato, mentre `print(m)` stampa il simbolo `None` (che può essere interpretato come "nessun valore"). Il motivo di ciò è che la chiamata della funzione **non restituisce nessun valore**, e quindi alla variabile `m` **non viene assegnato nessun valore**. Il messaggio `Buongiorno` viene solo **stampato** nella *shell*, ma non viene **restituito** come risultato della chiamata.

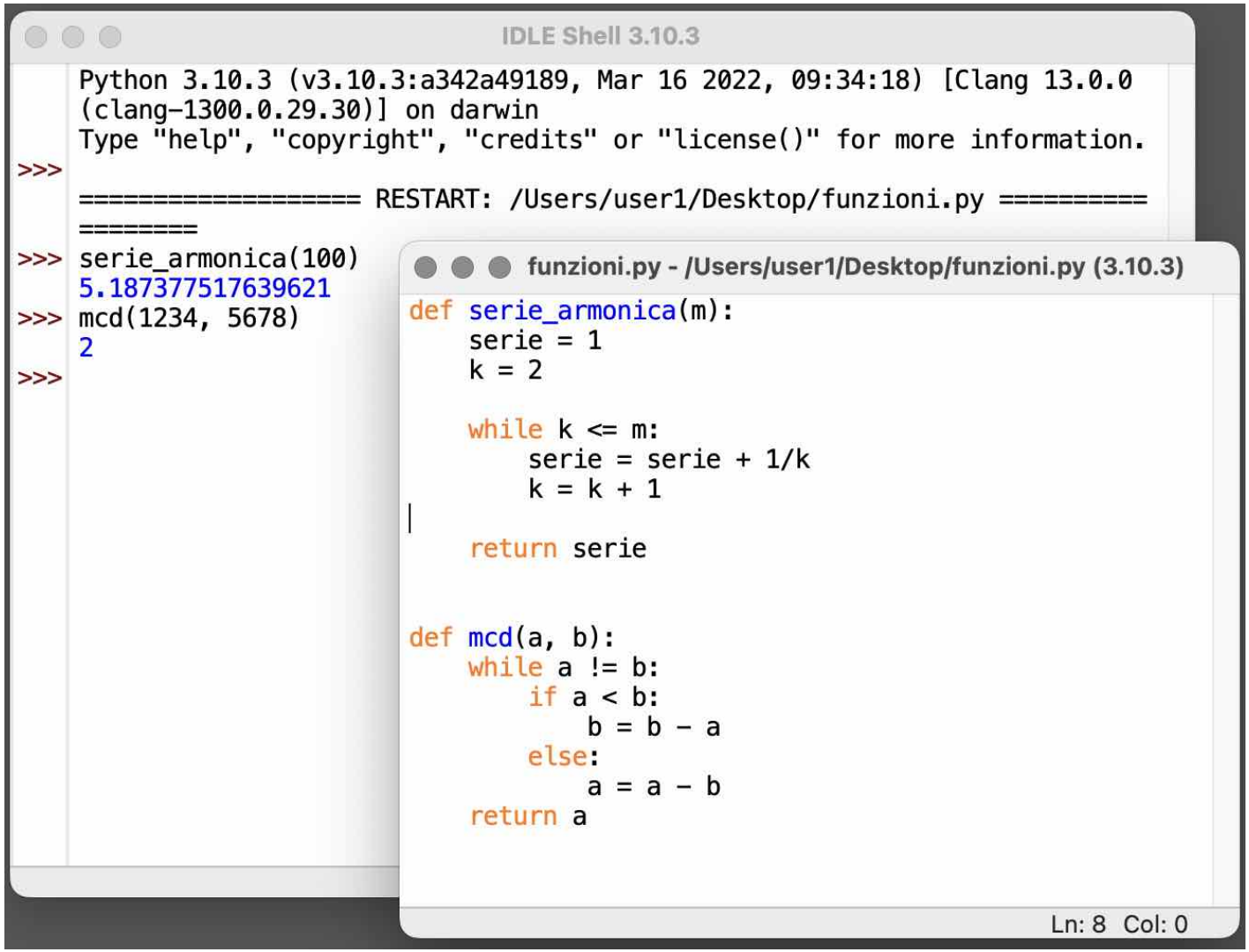


Definizione di più funzioni in uno stesso *file*

È anche possibile scrivere in uno stesso *file* la definizione di più funzioni, cioè una sequenza di istruzioni `def`.

Dopo aver eseguito il *file* (cioè, le istruzioni `def`), tutte le funzioni al suo interno saranno disponibili nella *shell*, come mostrato nell'esempio seguente.

Definizione di più funzioni in un *file*: esempio



Chiamate nidificate di funzioni

Nelle istruzioni del corpo di una funzione possono comparire chiamate (nidificate) di altre funzioni.

Se si vogliono scrivere chiamate nidificate di funzioni di libreria, sarà necessario eseguire **prima** la corrispondente istruzione `import`.

Di norma l'istruzione `import` viene inserita **all'inizio** di un *file* che contiene la definizione di nuove funzioni, **non** nel corpo di una di tai funzioni.

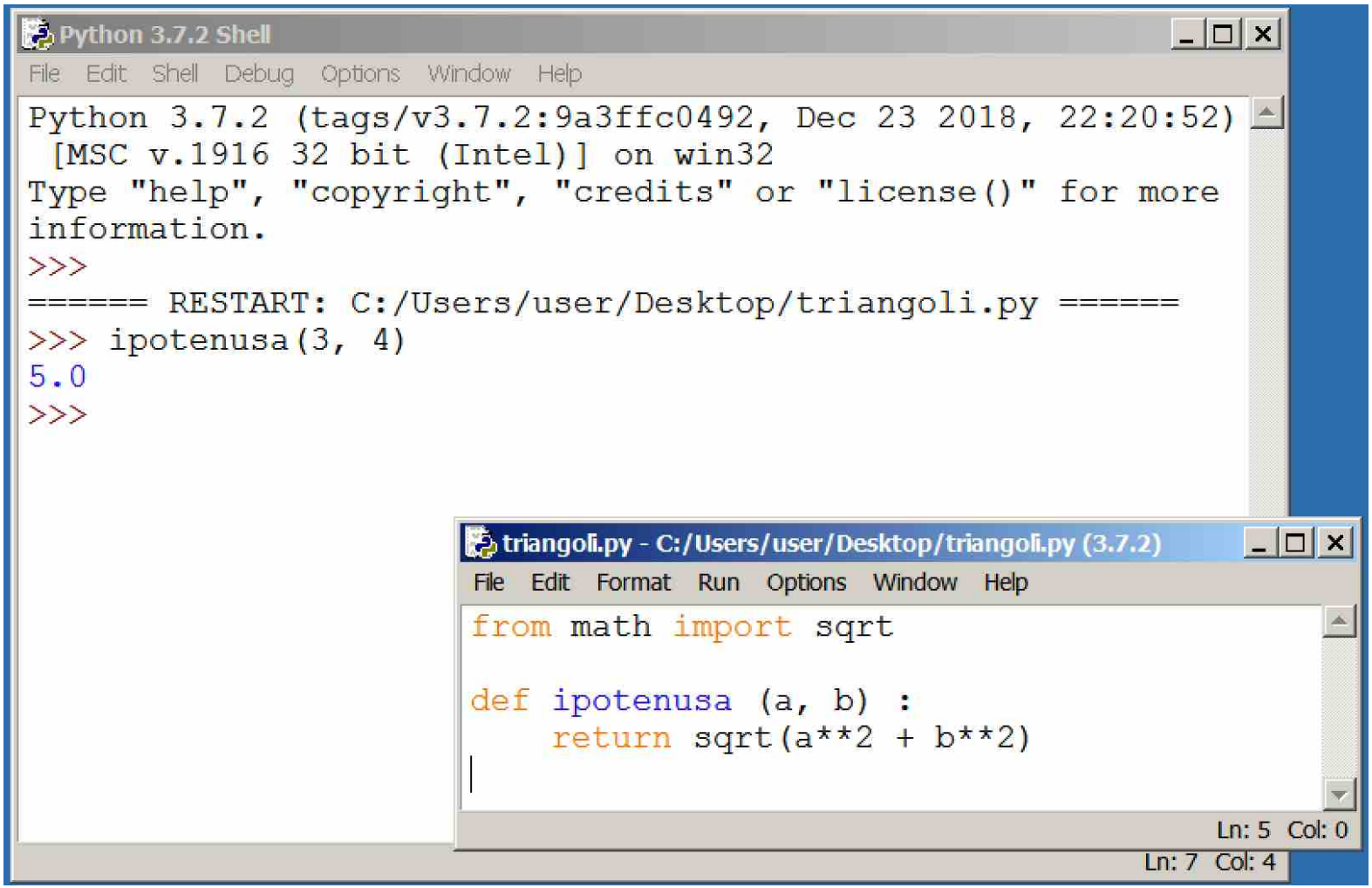
Chiamate nidificate di funzioni: esempio

In questo esempio si definisce una funzione che calcola la lunghezza dell'ipotenusa di un triangolo rettangolo, date le lunghezze dei cateti, usando la funzione `sqrt` definita nel modulo `math` (*file*: `36_ipotenusa_funzione.py`)

```
from math import sqrt

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

Chiamate nidificate di funzioni: esempio



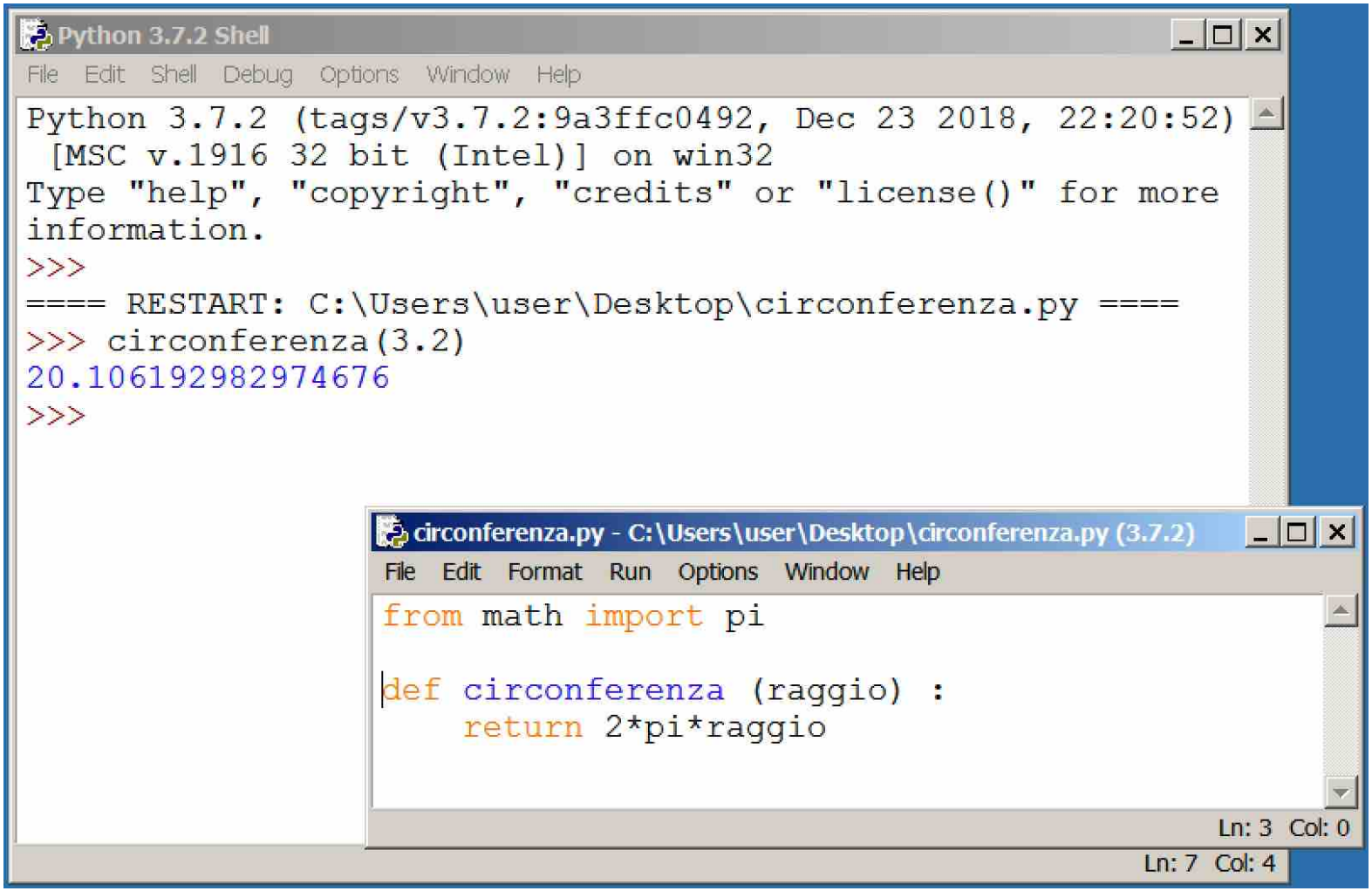
Chiamate nidificate di funzioni: esempio

La funzione seguente calcola la lunghezza della circonferenza di un cerchio, dato il suo raggio, usando la variabile `pi` definita nel modulo `math` (*file*: `37_circonferenza_funzione.py`)

```
from math import pi

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

Chiamate nidificate di funzioni: esempio



Chiamate nidificate di funzioni

Per poter chiamare dall'interno di una funzione *A* un'altra funzione *B* definita dall'utente, le alternative più semplici sono le seguenti

- la **definizione** di *A* e *B* deve trovarsi nello **stesso** *file*
- le due funzioni possono essere definite in *file* **diversi**, che però dovranno trovarsi in una **stessa** cartella; inoltre, nel *file* che contiene la chiamata di *B* si dovrà usare l'istruzione

  `from` nome-file `import` nome-funzione

  dove
  - nome-file è il nome del *file* che contiene la definizione di *B* (senza l'estensione `.py`)
  - nome-funzione è il nome della funzione *B*

Chiamate nidificate di funzioni: esempio

In precedenza si è visto un programma che stampa nella *shell* tutti i numeri primi fino a un dato intero n, il cui valore viene acquisito attraverso la tastiera (*file*: `31_sequenza_numeri_primi_1.py`). In questo esempio si mostra come realizzare la stessa operazione attraverso due funzioni.

La prima funzione (`numero_primo`) riceve come argomento un numero naturale e determina se questo sia primo, restituendo il valore `True` oppure `False`.

La seconda (`stampa_numeri_primi`) riceve come argomento il valore di n, genera tutti gli interi da 1 a n mediante un'istruzione iterativa, e per ciascuno di essi chiama la funzione `numero_primo`, stampando solo i numeri che risultano essere primi.

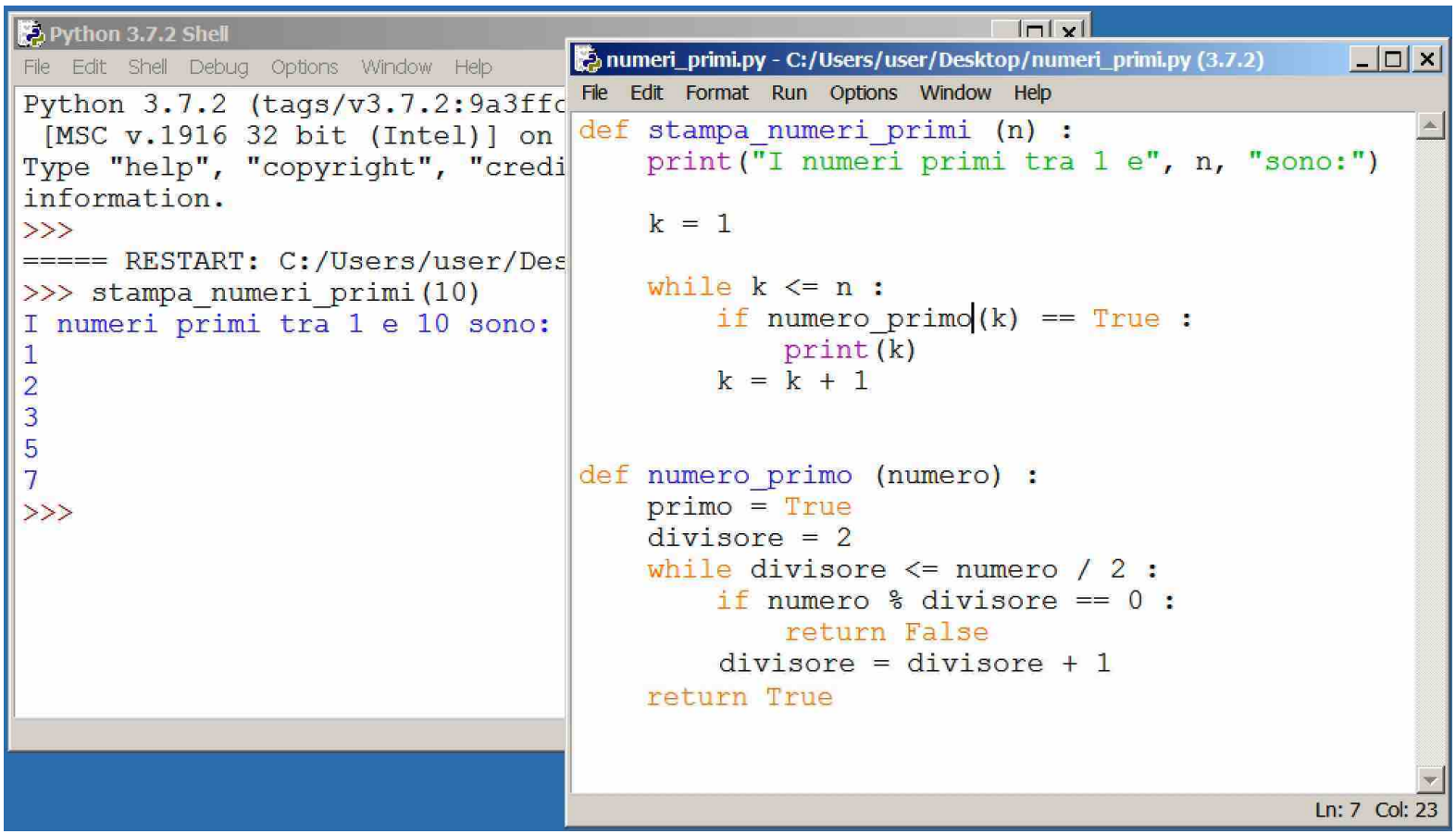
Chiamate nidificate di funzioni: esempio

```
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

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

Chiamate nidificate di funzioni: esempio

Nella prima versione (*file*: `38_sequenza_numeri_primi_funzione.py`) le due funzioni vengono definite in uno stesso *file*:



Chiamate nidificate di funzioni: esempio

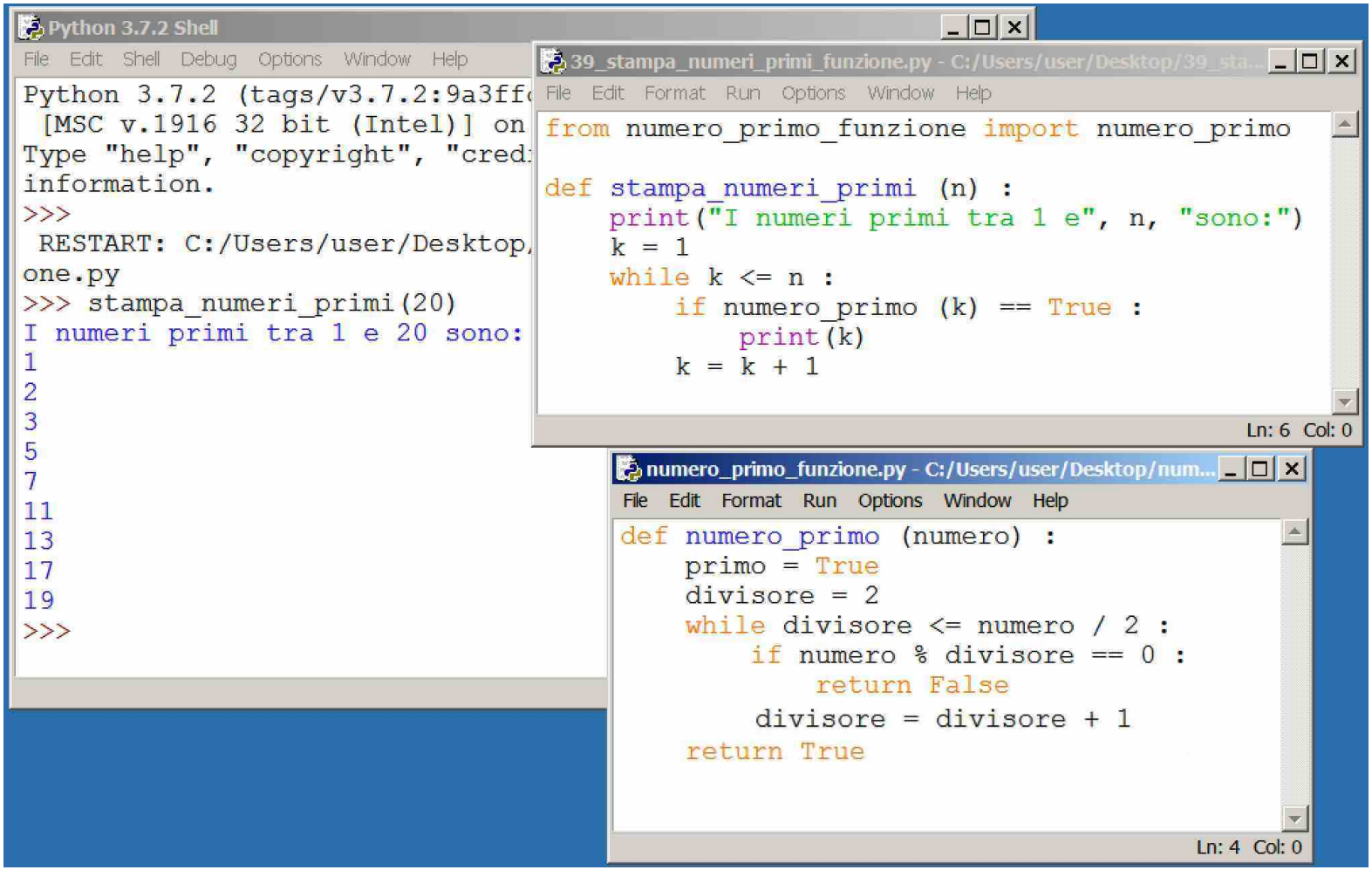
Nella seconda versione (*file*: `39_stampa_numeri_primi_funzione.py` e `numero_primo_funzione.py`) le funzioni vengono definite in due *file* diversi, che devono essere memorizzati in una stessa cartella.

Il *file* che contiene la **definizione** della funzione `stampa_numeri_primi` deve anche contenere l'istruzione:

```
from numero_primo_funzione import numero_primo
```

Prima di poter chiamare le funzioni bisogna eseguire solo le istruzioni di quest'ultimo *file*, poiché la definizione della funzione contenuta nell'altro *file* viene eseguita come conseguenza dell'istruzione `import`.

Chiamate nidificate di funzioni: esempio



Programmi che contengono definizioni di funzioni

In un *file* è anche possibile scrivere un programma che contenga sia definizioni di nuove funzioni, che sequenze d'istruzioni che chiamano tali funzioni.

L'unico vincolo è che l'esecuzione di ogni chiamata può avvenire solo **dopo** l'esecuzione della definizione (ovvero dell'istruzione `def`) della funzione corrispondente.

Di norma, tutte le definizioni di nuove funzioni vengono scritte **all'inizio** di un *file*.

Esempio

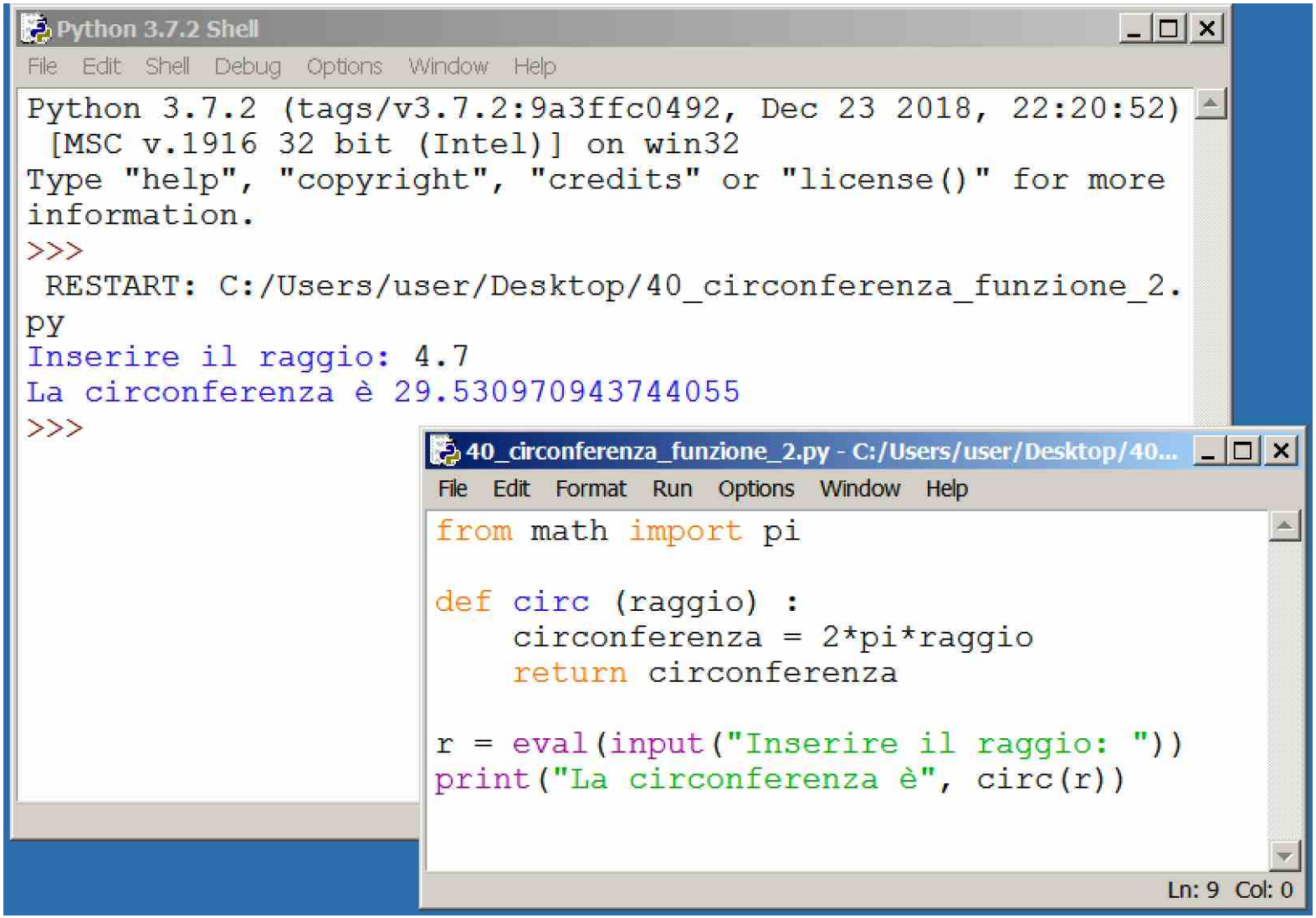
Nel programma mostrato in basso si definisce la funzione `circ` già vista in precedenza; le istruzioni successive acquisiscono il valore del raggio e stampano la lunghezza della circonferenza corrispondente, mediante una chiamata della funzione `circ`
(*file*: `40_circonferenza_funzione_2.py`)

```
from math import pi

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

r = input("Inserire il raggio: ")
print("La circonferenza è", circ(r))
```

Esempio


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
RESTART: C:/Users/user/Desktop/40_circonferenza_funzione_2.
py
Inserire il raggio: 4.7
La circonferenza è 29.530970943744055
>>>
40_circonferenza_funzione_2.py - C:/Users/user/Desktop/40...
File Edit Format Run Options Window Help
from math import pi
def circ (raggio) :
circonferenza = 2*pi*raggio
return circonferenza
r = eval(input("Inserire il raggio: "))
print("La circonferenza è", circ(r))
Ln: 9 Col: 0

Funzioni: variabili *locali*

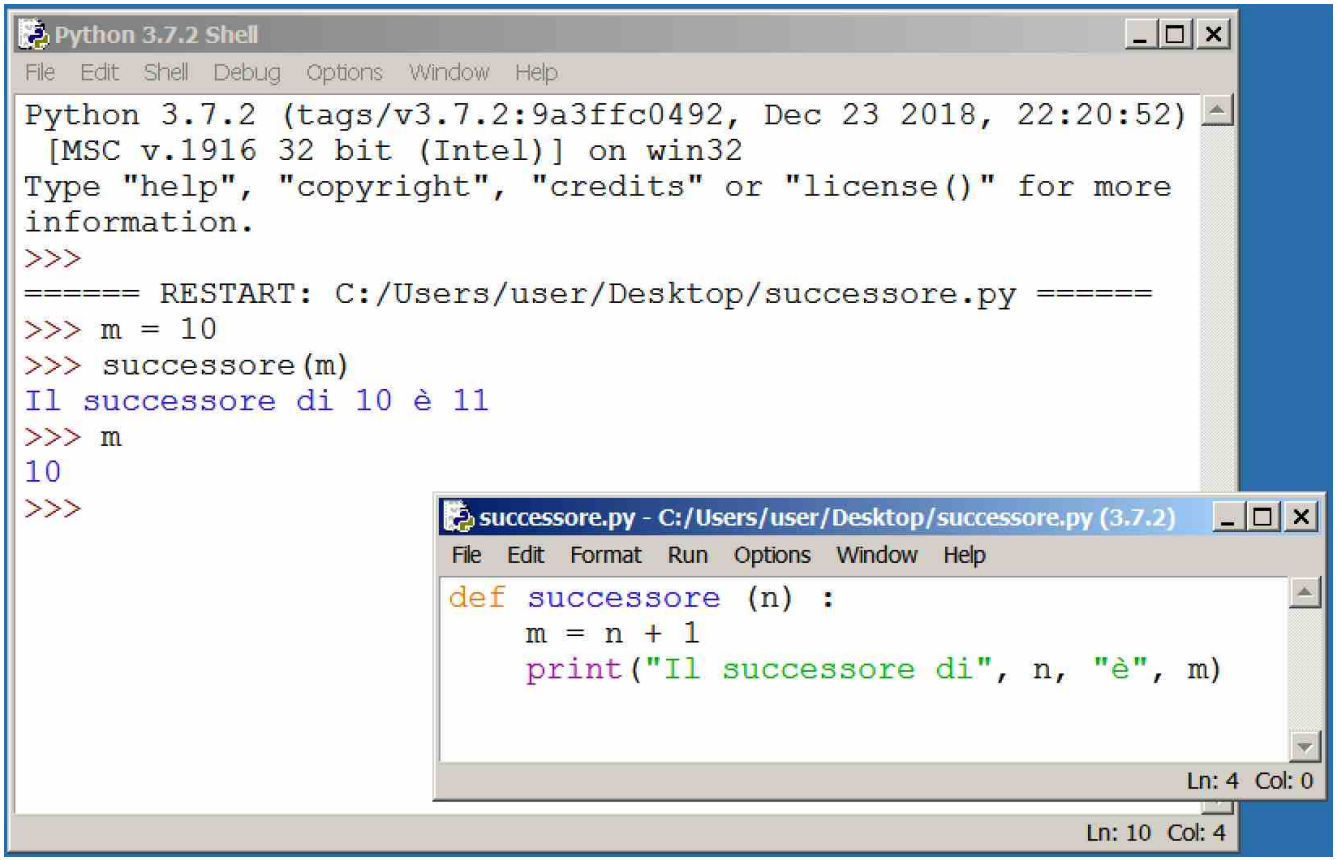
I parametri di una funzione e le eventuali altre variabili alle quali viene **assegnato** un valore all'interno di essa sono dette **locali**, in quanto esse vengono "create" dall'interprete nel momento in cui la funzione viene eseguita (mediante una chiamata), e vengono "eliminate" quando l'esecuzione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

In particolare, se la chiamata della funzione viene scritta nella *shell*, e prima della chiamata è stata definita nella stessa *shell* una variabile con lo stesso nome di una delle variabili della funzione

- le due variabili vengono associate a celle di memoria **distinte**
- durante l'esecuzione della funzione l'interprete potrà accedere solo alla variabile associata alla funzione, e non potrà quindi usare o modificare il valore associato alla variabile avente lo stesso nome definita nella *shell*, che manterrà il valore originale

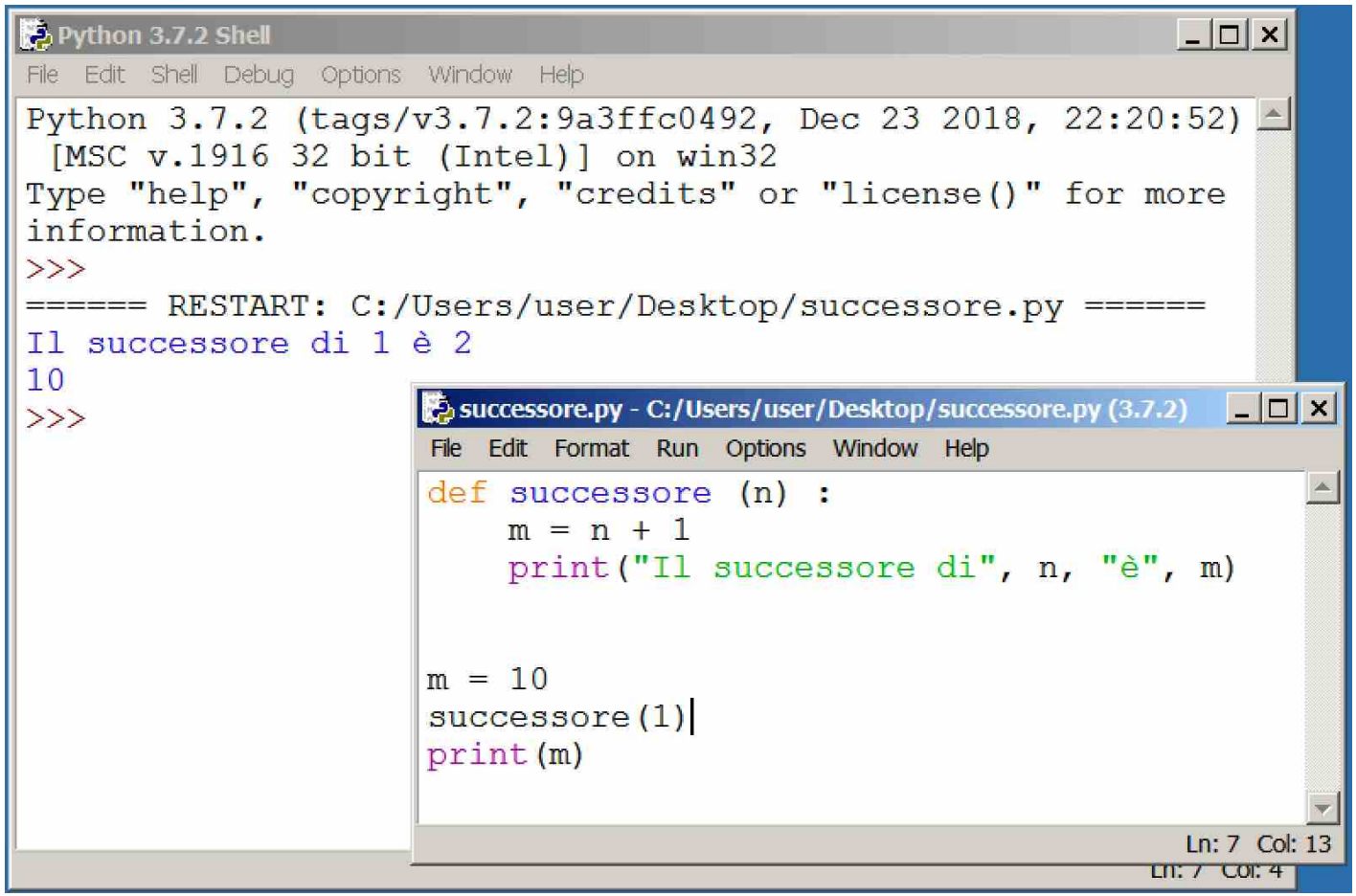
Variabili locali: esempio

Questo esempio mostra che il valore associato alla variabile *n*, definita nella *shell* **prima** della chiamata della funzione `successore`, non viene modificato durante l'esecuzione della funzione dall'istruzione `n = x + 1` (notare che la funzione non **restituisce** nessun valore)



Variabili locali: esempio

Lo stesso accade se la chiamata della funzione si trova nelle istruzioni di un programma scritto in un *file*, rispetto alle variabili definite nello stesso programma, come mostra l'esempio seguente



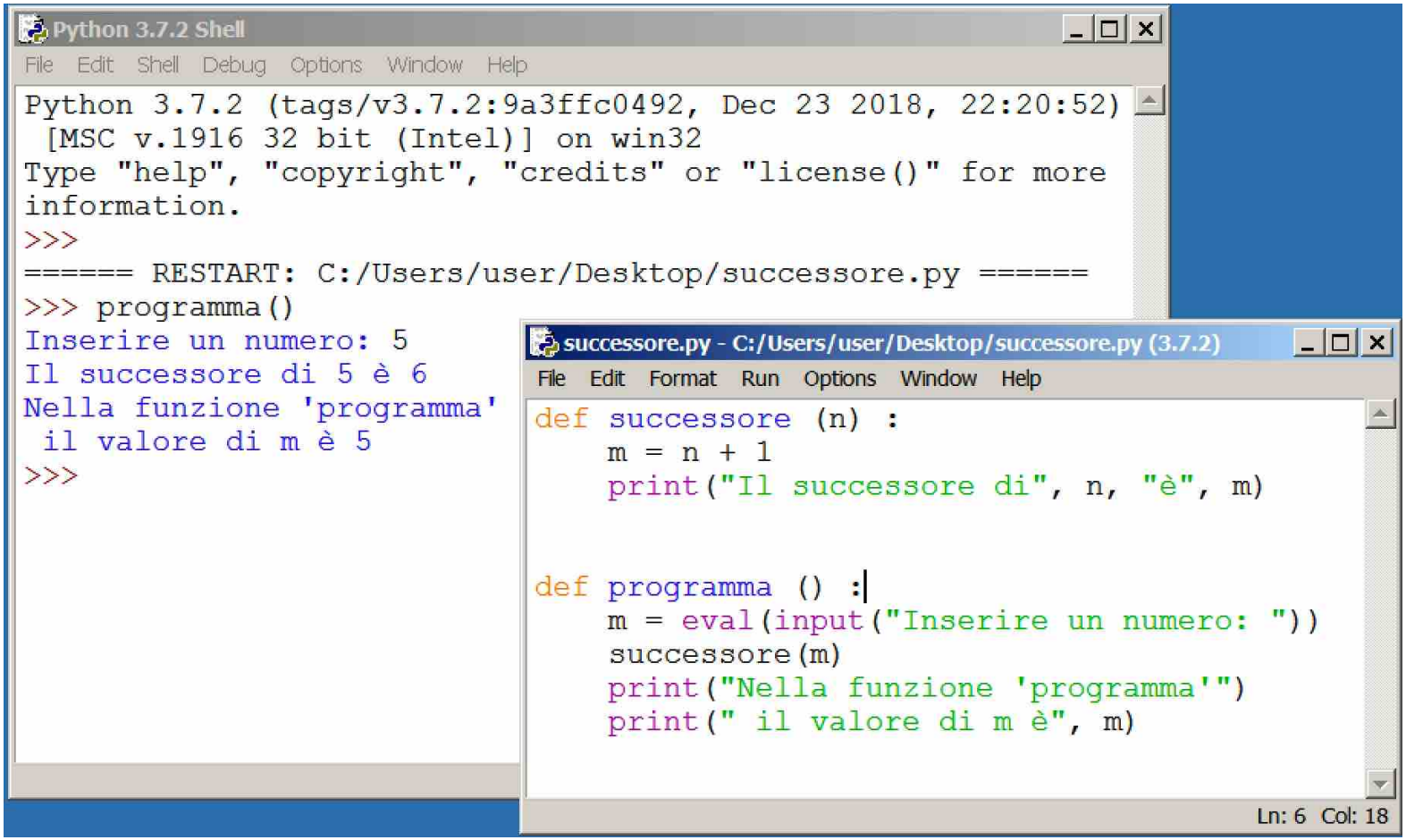
Variabili locali

Anche le variabili definite in funzioni **diverse** sono variabili locali.

Ciò significa che se una funzione B è chiamata da istruzioni che si trovano nel corpo di un'altra funzione A, sia A che B potranno accedere solo alle proprie variabili, ma non potranno accedere a quelle dell'altra funzione, né potranno quindi modificarle.

Questo consente in particolare di usare, in funzioni diverse, variabili aventi lo stesso nome. Un esempio è mostrato di seguito.

Variabili locali: esempio



Variabili locali

Il fatto che le variabili definite in una funzione siano locali consente di definire nuove funzioni senza preoccuparsi dell'eventuale presenza di variabili con lo stesso nome nei programmi che le useranno.

Analogamente, quando si scrive un programma che chiama funzioni predefinite, di libreria, o definite dall'utente, non ci si deve preoccupare dei nomi dei parametri e delle eventuali variabili definite in tali funzioni.

Funzioni: variabili *globali*

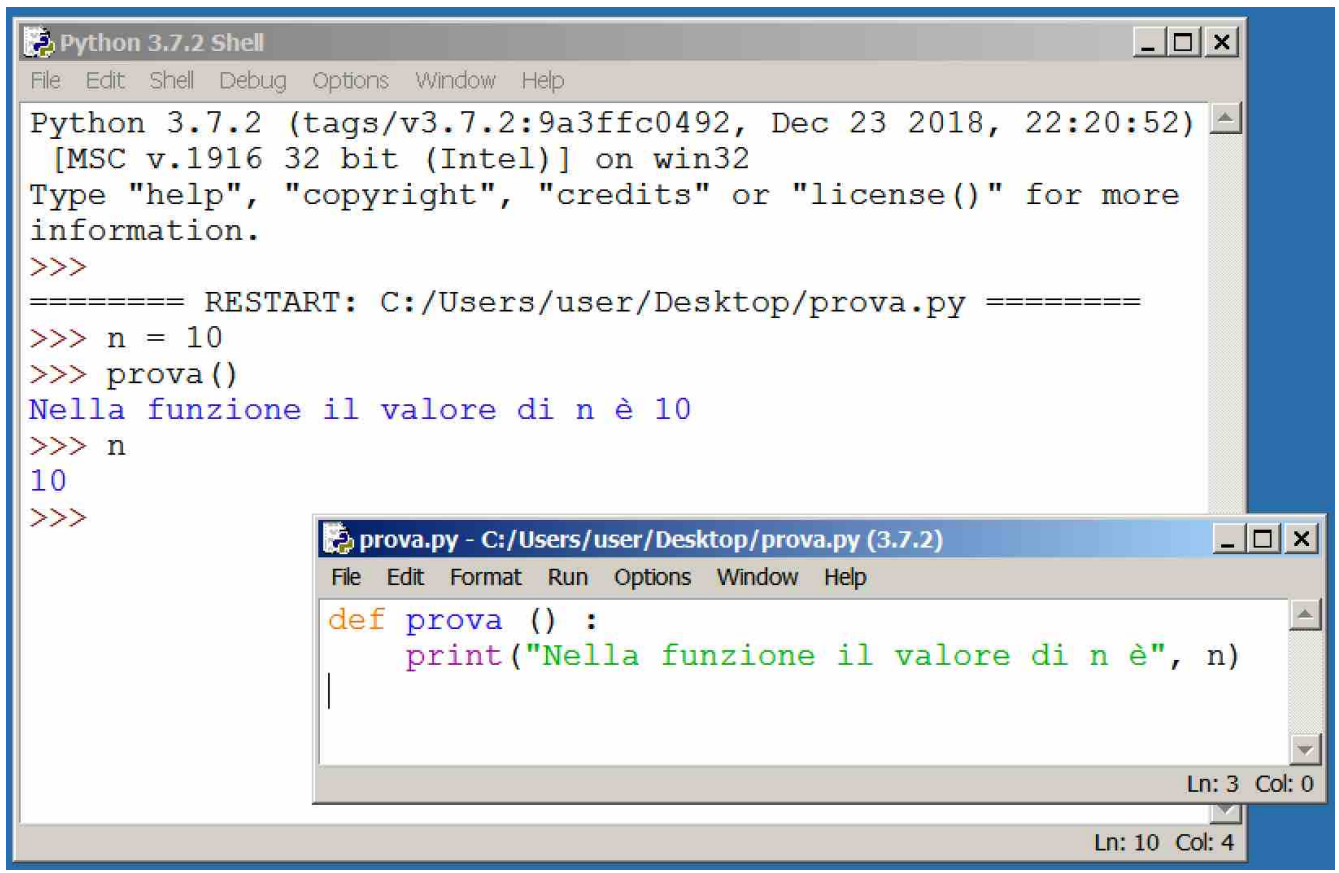
Se invece all'interno di una funzione il nome di una variabile (che non sia uno dei parametri) compare in un'espressione senza che in precedenza nella funzione sia stato **assegnato** a essa alcun valore, tale variabile è considerata **globale**, cioè l'interprete assume che il suo valore sia stato definito nelle istruzioni precedenti la **chiamata** della funzione (scritte nella *shell* o in un programma); se ciò non è avvenuto, si otterrà un messaggio d'errore.

In questo modo le istruzioni di una funzione possono accede al valore associato a variabili definite nella *shell* o nel programma chiamante.

Tuttavia è preferibile **evitare** l'uso di variabili globali nelle funzioni, poiché la loro presenza rende più difficile assicurare la correttezza di un programma e comprenderne il funzionamento.

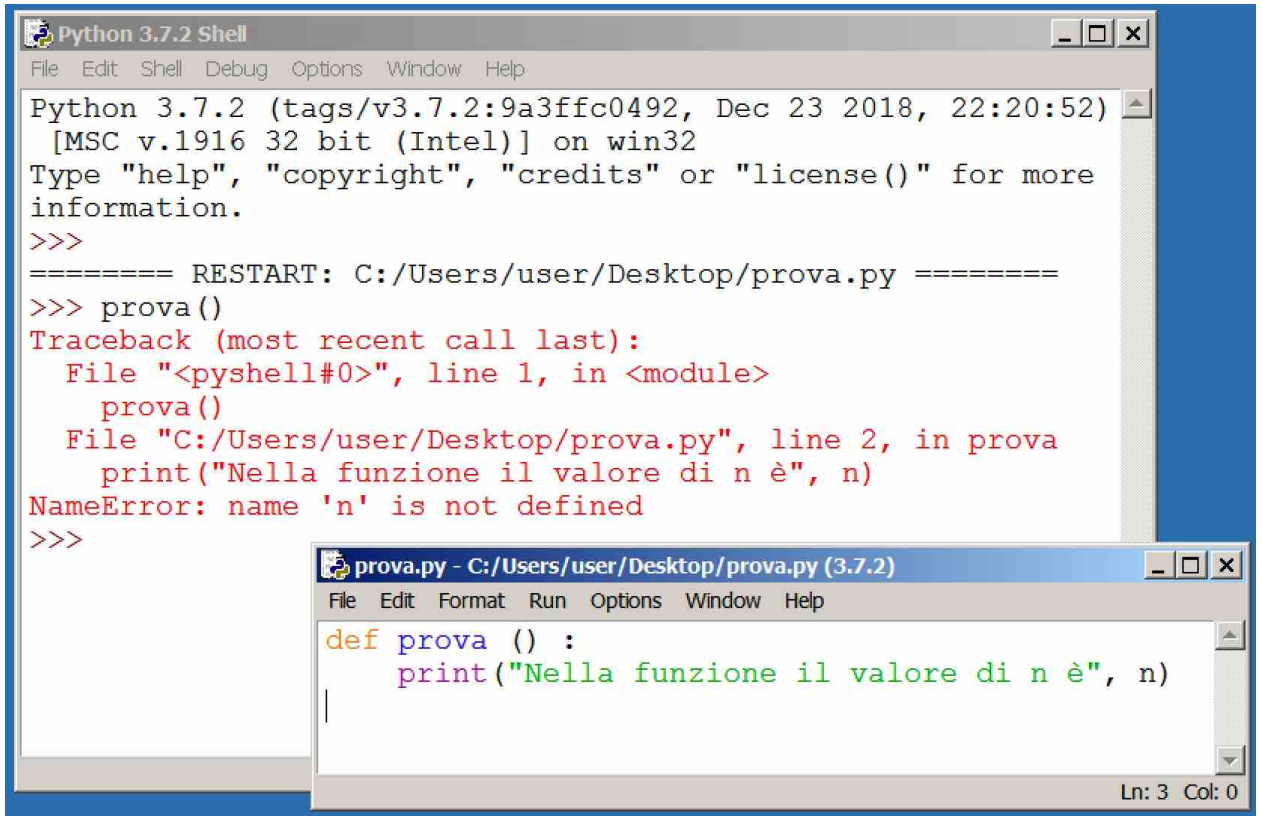
Variabili globali: esempio

Nella funzione `prova` si fa riferimento al valore associato alla variabile `n` senza che a essa sia stato assegnato in precedenza alcun valore all'interno della stessa funzione: in questo caso l'interprete accede alla variabile `n` già definita nella `shell`.



Variabili globali: esempio

Ovviamente, se nella `shell` **non** fosse già stata definita una variabile di nome `n`, si otterrebbe un messaggio d'errore durante l'esecuzione della funzione



Strutturazione dei programmi

Nei linguaggi di alto livello la possibilità di definire nuove funzioni consente anche di **strutturare** un programma. ovvero suddividerlo in diversi "sottoprogrammi" (funzioni), ciascuno dei quali esegue un'operazione distinta e indipendente dagli altri.
Questo stile di programmazione è detto **modulare**.

In particolare, è buona norma cercare di suddividere un programma in funzioni che siano il più possibile **brevi**.

La programmazione modulare presenta diversi vantaggi

- semplifica la scrittura, la comprensione e la modifica dei programmi
- rende più facile l'individuazione di eventuali errori
- consente di usare una **stessa** funzione in programmi **diversi**

Strutturazione dei programmi

Un programma Python può essere strutturato suddividendolo in una o più funzioni e in una sequenza d'istruzioni che costituirà il programma "principale".

Le funzioni e le istruzioni del programma "principale" potranno trovarsi in uno o più *file*. Nel caso di più *file* si dovranno prevedere opportune istruzioni `import`.

Per eseguire il programma si dovrà eseguire il *file* che contiene le istruzioni del programma "principale".

In alternativa, anche le istruzioni del programma "principale" potranno essere scritte sotto forma di funzione: per avviare l'intero programma si dovrà chiamare tale funzione dalla *shell*.

Strutturazione dei programmi: esempio

Si vuole scrivere un programma che acquisisca un numero naturale n e stampi tutti i numeri primi compresi tra 1 e n.

In precedenza si sono definite due funzioni per verificare se un numero sia primo, e per stampare tutti i numeri primi in un certo intervallo (*file*: `38_sequenza_numeri_primi_funzione.py`).

Di seguito si mostra come l'intero programma (compresa l'acquisizione del valore di n) possa essere strutturato nei due modi sopra descritti.

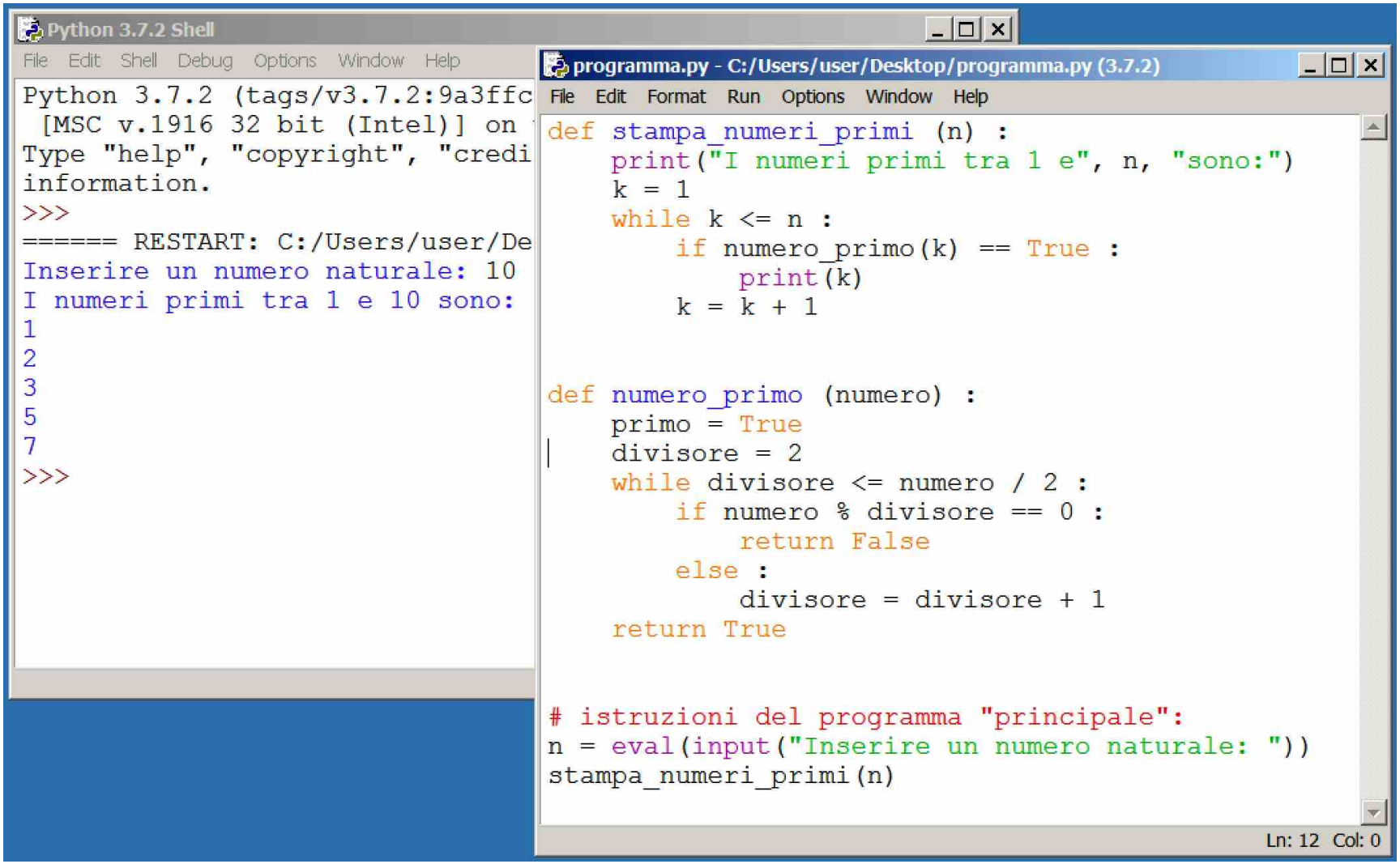
Strutturazione dei programmi: esempio

Nel *file* `41_sequenza_numeri_primi_programma_1.py` alle due funzioni si aggiungono le istruzioni che acquisiscono il valore di n e chiamano la funzione `stampa_numeri_primi`:

```
n = input("Inserire un numero naturale: ")
stampa_numeri_primi(n)
```

Il programma può essere avviato eseguendo il *file* che lo contiene.

Strutturazione dei programmi: esempio



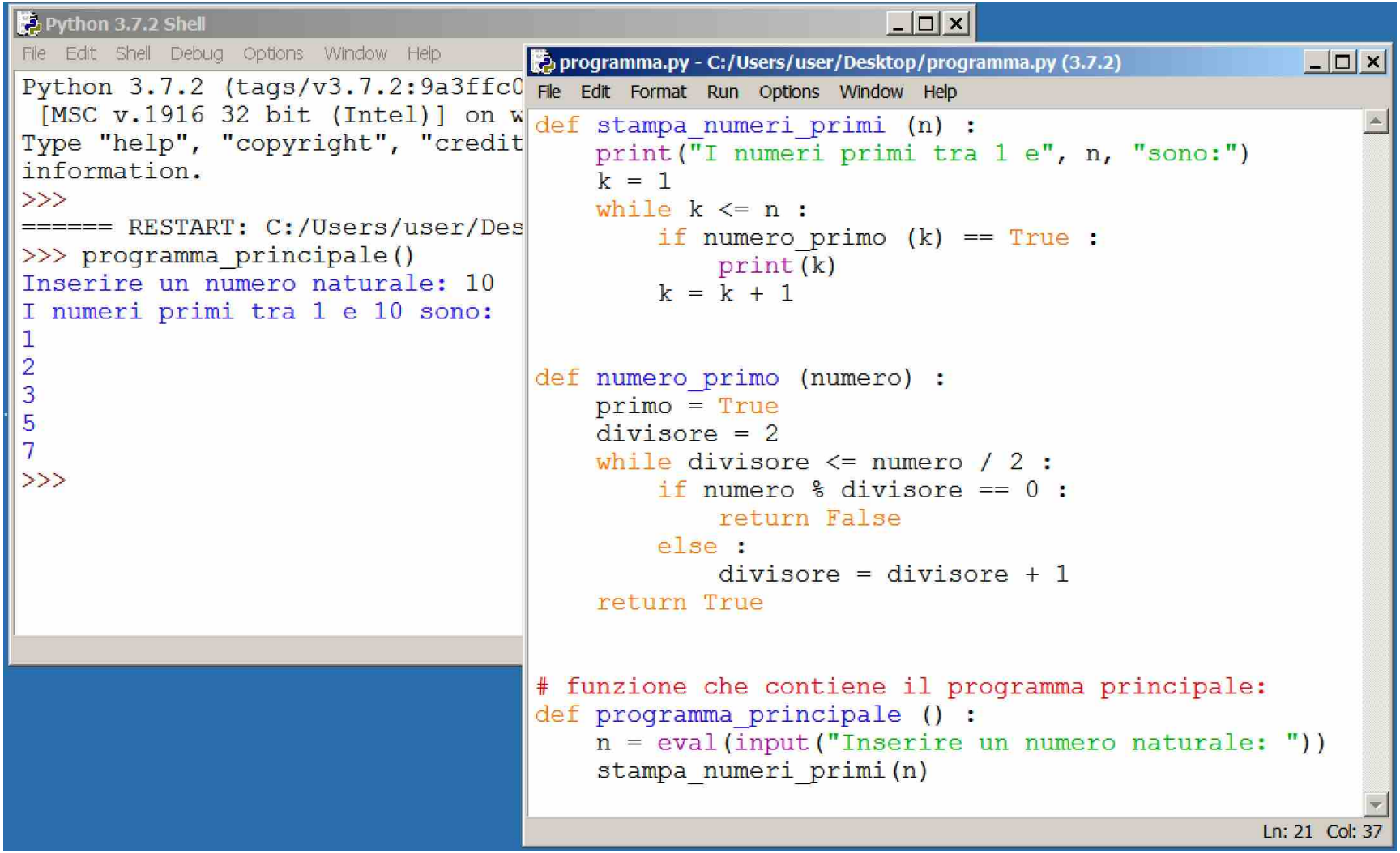
Strutturazione dei programmi: esempio

Nel *file* `42_sequenza_numeri_primi_programma_2.py` anche le istruzioni che acquisiscono il valore di n e chiamano la funzione `stampa_numeri_primi` sono inserite in una funzione:

```
def programma_principale():
    n = eval(input("Inserire un numero naturale: "))
    stampa_numeri_primi(n)
```

In questo caso per eseguire il programma si dovrà chiamare dalla *shell* la funzione `programma_principale` (dopo aver eseguito le istruzioni `def`).

Strutturazione dei programmi: esempio



Tipi dai dato strutturati

Tipi di dato nei linguaggi di alto livello

I valori che possono essere rappresentati ed elaborati nei programmi scritti in linguaggi di alto livello vengono suddivisi in diversi **tipi di dato**.

Per *tipo di dato* s'intende

- un insieme di **valori**
- un insieme di **operazioni** eseguibili su tali valori

Abbiamo già incontrato alcuni tra i principali tipi di dato del linguaggio Python: numeri interi, numeri frazionari, stringhe, valori logici.

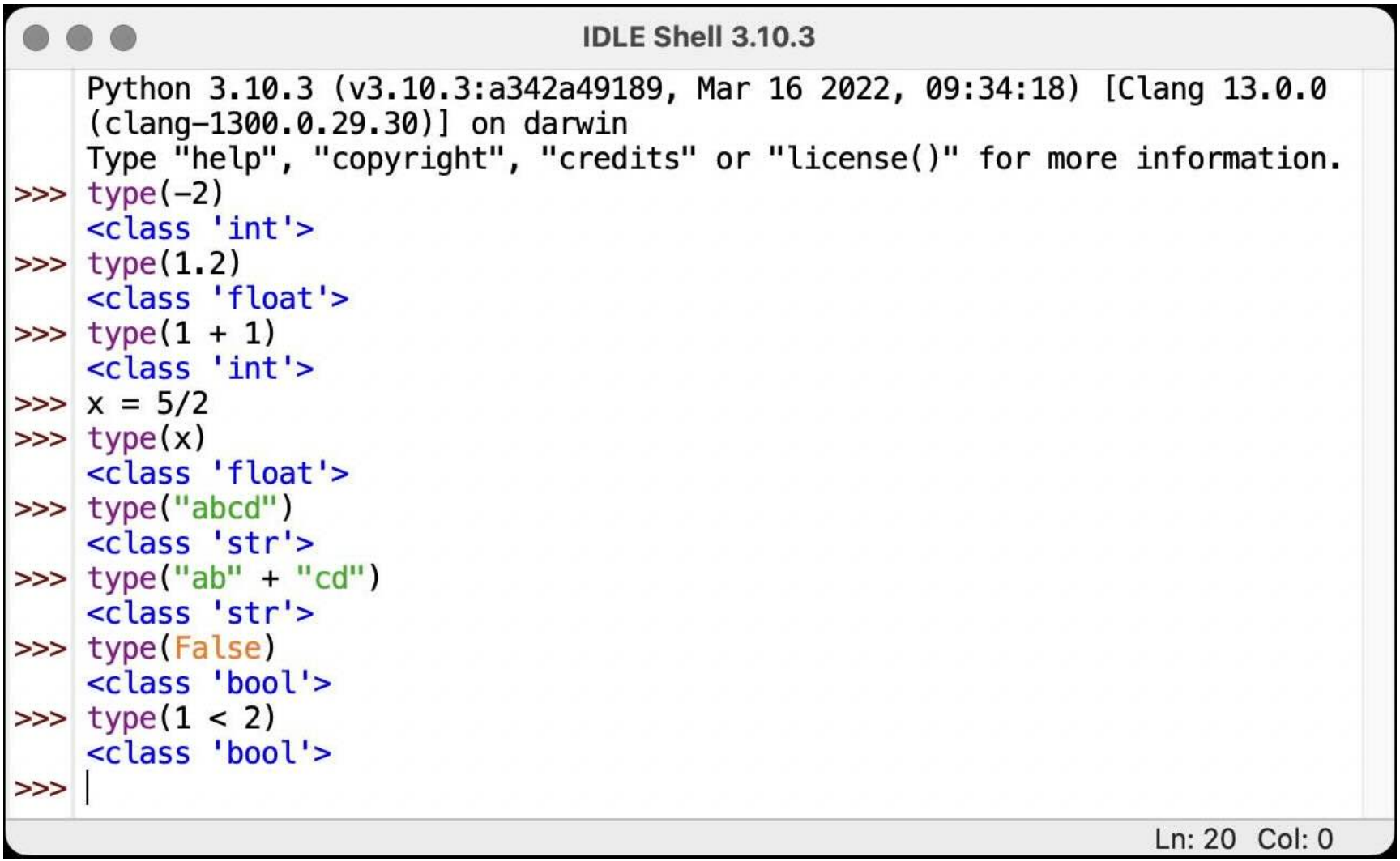
Tipi di dato Python: la funzione `type`

Nel linguaggio Python ogni tipo di dato è associato a una **classe** (concetto legato alla *programmazione a oggetti*, non considerata in questo corso) e a un nome simbolico (identificatore); per esempio

- numeri interi: `int`
- numeri frazionari: `float`
- stringhe: `str`
- valori logici ("booleani"): `bool`

La funzione predefinita `type` restituisce l'identificatore associato al tipo di dato del suo argomento, che può essere una qualsiasi espressione Python, come mostrato negli esempi seguenti.

La funzione `type`: esempi



Tipi di dato Python: esempi

Il tipo di dato "numero intero" (`int`) è definito come

- l'insieme dei numeri interi che possono essere codificati in complemento a due con n bit (dove n dipende dall'architettura dello specifico calcolatore): $\{-2^{n-1}, \ldots, +2^{n-1} - 1\}$
- un insieme di operatori, inclusi i seguenti
  - aritmetici: `+ - * / % **`
  - di confronto: `== != < <= => >`

Il tipo di dato "stringa" (`str`) è definito come

- l'insieme delle sequenze di zero o più caratteri (senza un limite superiore predefinito sulla lunghezza), rappresentati tra apici singoli o doppi
- un insieme di operatori, tra i quali la concatenazione (+) e gli operatori di confronto: `== != < <= => >`

Tipi semplici e strutturati

I tipi di dato vengono classificati a loro volta in

- tipi **semplici**
- tipi **strutturati**

Un tipo **semplice** è composto da valori che non possono essere "scomposti" in valori più semplici ai quali sia possibile accedere attraverso operatori o funzioni del linguaggio.
Esempi di tipi semplici del linguaggio Python (e di altri linguaggi) sono i numeri interi, i numeri frazionari e i valori logici.

Un tipo **strutturato** è invece composto da valori che sono a loro volta collezioni o sequenze di valori più semplici.
Le stringhe sono un esempio di tipo strutturato: sono infatti composte da **sequenze ordinate** di caratteri a ciascuno dei quali è possibile accedere **individualmente**, come si vedrà più avanti.

Tipi di dato strutturati del linguaggio Python

Oltre alle stringhe, in questo corso vengono considerati due tipi strutturati del linguaggio Python

- **liste**: **sequenze ordinate** di valori qualsiasi
- **dizionari**: **collezioni** (non ordinate) di valori qualsiasi

Il tipo di dato *lista*

In molte applicazioni i dati da elaborare sono costituiti da **sequenze ordinate** di valori.

Per esempio, le coordinate di un punto o gli elementi di un vettore in un dato sistema di riferimento possono essere rappresentati come una sequenza ordinata di numeri reali.

Nei programmi è conveniente poter rappresentare dati di questa natura come un **singolo** valore **composto**, che possa essere assegnato a una **singola** variabile.

Come si è già visto, nel caso particolare delle sequenze di caratteri questo è possibile mediante il tipo di dato *stringa*.

Il tipo **lista** fornisce questa possibilità per sequenze ordinate di valori **qualsiasi**, anche eterogenei (cioè di tipi diversi).

Il tipo di dato *lista*

Una **lista** si rappresenta in un programma Python come

- una sequenza ordinata di valori
- scritti tra parentesi **quadre**
- separati da virgole

Esempi

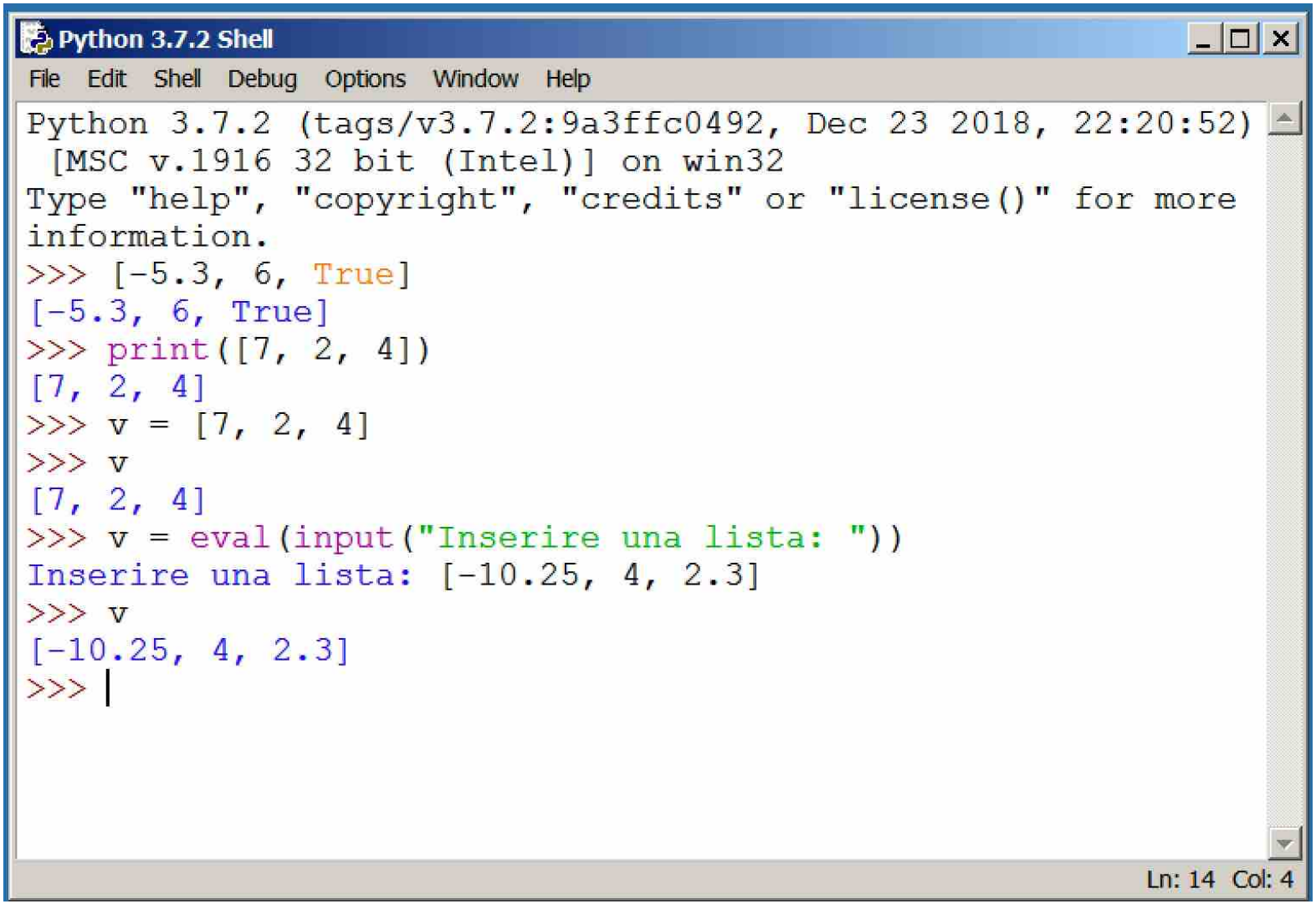
- `[7, -2, 4]`
  una lista composta da tre numeri interi
- `[-5.3, 6, True]`
  una lista composta da un numero reale, un numero intero e un valore logico
- `[]`
  una lista **vuota**

Il tipo di dato *lista*

Come per tutti i tipi di dato del linguaggio Python, la rappresentazione di una lista è un'**espressione**. È quindi possibile

- scrivere una lista nella *shell*: dopo la pressione del tasto `INVIO`, la lista verrà stampata nella stessa *shell*, così come accade per i valori degli altri tipi (numeri, ecc.)
- stampare una lista mediante l'istruzione `print`, per esempio:
  `print([7, -2, 4])`
- assegnare una lista a una variabile, per esempio:
  `v = [7, -2, 4]`
- acquisire una lista attraverso la tastiera, per esempio:
  `v = eval(input("Inserire una lista: "))`

Esempi


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>> [-5.3, 6, True]
[-5.3, 6, True]
>>> print([7, 2, 4])
[7, 2, 4]
>>> v = [7, 2, 4]
>>> v
[7, 2, 4]
>>> v = eval(input("Inserire una lista: "))
Inserire una lista: [-10.25, 4, 2.3]
>>> v
[-10.25, 4, 2.3]
>>>
Ln: 14 Col: 4

Il tipo di dato *lista*

In generale ogni elemento di una lista è costituito dal valore di un'**espressione**.

Per esempio, dopo l'esecuzione delle istruzioni

```
x = 2
y = -5
z = [x, y**2 + 1, x == 3]
```

la variabile `z` sarà associata alla lista `[2, 26, False]`

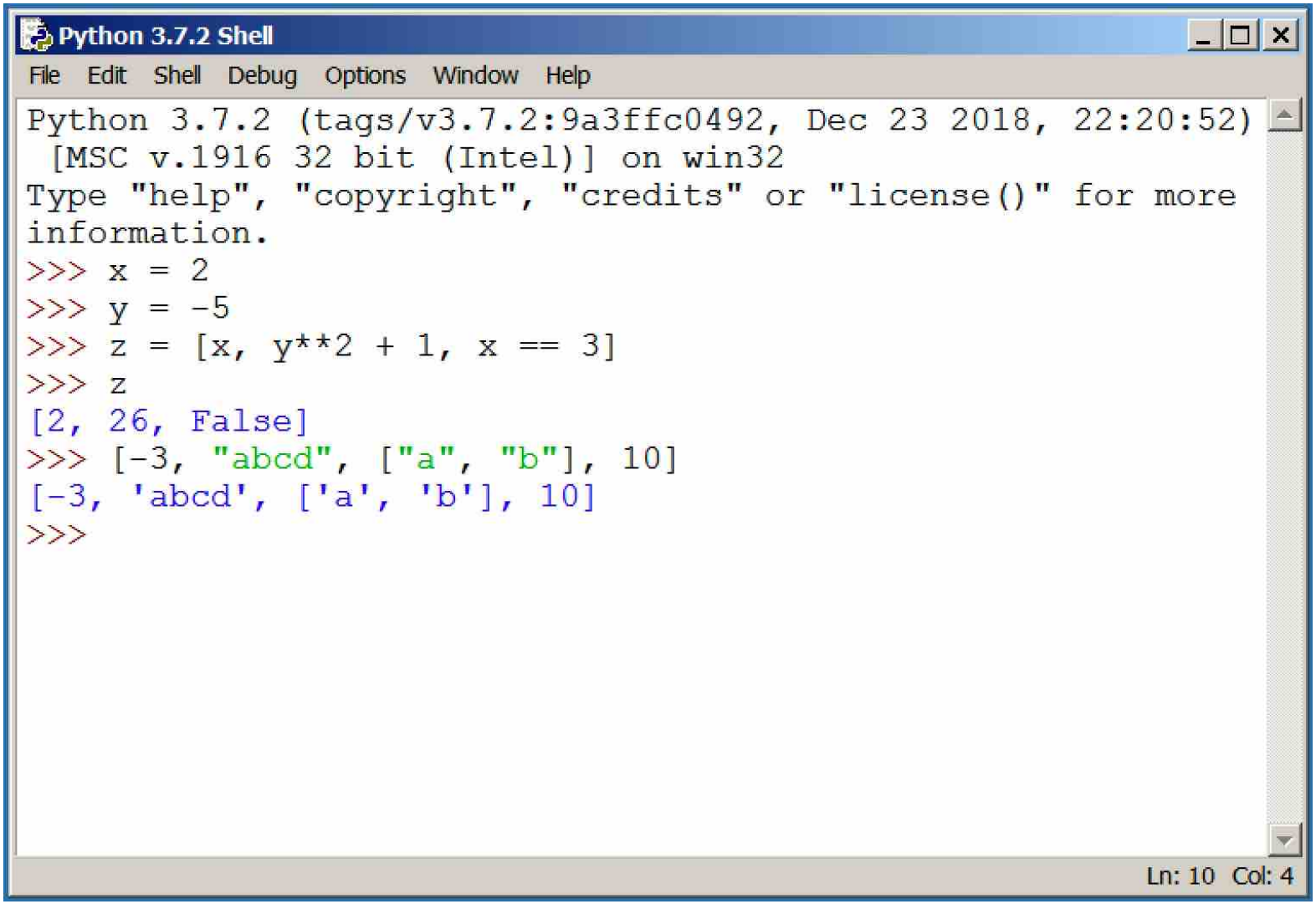
Il tipo di dato *lista*

Poiché gli elementi di una lista possono essere valori di qualsiasi tipo, essi possono a loro volta essere liste: si parla in questo caso di liste **nidificate**.

Per esempio, la seguente espressione produce come risultato una lista di **quattro** elementi: un numero intero, una stringa, una lista di due elementi (strighe), e un altro numero intero:

```
[-3, "abcd", ["a", "b"], 10]
```

Esempi


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>> x = 2
>>> y = -5
>>> z = [x, y**2 + 1, x == 3]
>>> z
[2, 26, False]
>>> [-3, "abcd", ["a", "b"], 10]
[-3, 'abcd', ['a', 'b'], 10]
>>>
Ln: 10 Col: 4

Il tipo di dato *lista*

Anche per il tipo di dato *lista* il linguaggio Python prevede diversi operatori e funzioni predefinite.

In particolare, essendo le liste un tipo strutturato, alcuni operatori consentono l'accesso ai **singoli** elementi di una lista.

Il meccanismo di accesso si basa sul fatto che ogni elemento è identificato univocamente dalla sua **posizione** (si ricordi che una lista è una sequenza **ordinata** di valori).

La posizione di ciascun elemento di una lista è rappresentata in linguaggio Python attraverso un numero intero, detto **indice**. Per convenzione, l'indice del primo elemento di una lista è 0, l'indice del secondo elemento è 1, e così via.

Principali operatori sulle liste

La sintassi degli operatori principali è riassunta in tabella, ed è descritta di seguito in maggior dettaglio, insieme alla semantica.
Si noti che ciascun operatore consente di scrivere una **espressione**.

sintassi	**descrizione**
lista$_1$ == lista$_2$	confronto ("uguale a")
lista$_1$!= lista$_2$	confronto ("diverso da")
espressione `in` lista	verifica della presenza di un valore in una lista
espressione `not in` lista	verifica dell'assenza di un valore in una lista
lista$_1$ + lista$_2$	concatenazione
lista`[indice]`	indicizzazione: accesso a un elemento
lista`[`indice$_1$`:`indice$_2$`]`	*slicing* (sezionamento): accesso a una sotto-sequenza di elementi

Operatori di confronto

Gli operatori == e != consentono di scrivere espressioni **condizionali** (il cui valore sarà `True` o `False`) consistenti nel confronto tra due liste.

Sintassi

- lista_1 == lista_2
- lista_1 != lista_2

dove lista_1 e lista_2 indicano **espressioni** che abbiano come valore una lista.

Semantica: due liste sono considerate identiche se sono composte dallo stesso numero di elementi, e se ogni elemento ha valore identico a quello dell'elemento che si trova nella **stessa posizione** nell'altra lista.

Gli operatori `in` e `not in`

Questi operatori consentono di scrivere espressioni **condizionali** che hanno lo scopo di verificare se un certo valore sia presente o meno all'interno di una lista.

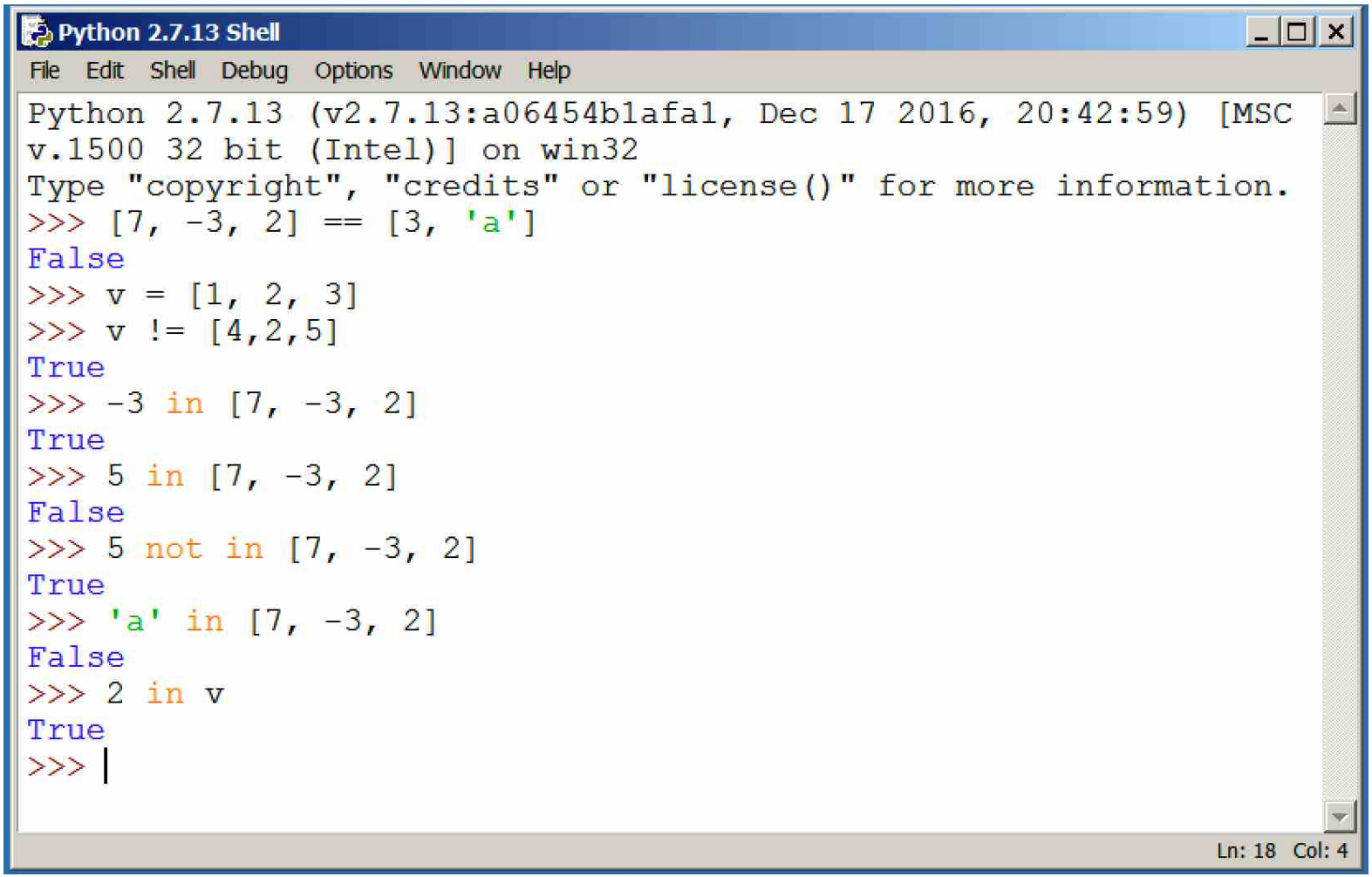
Sintassi

- espressione `in` lista
- espressione `not in` lista

dove espressione indica una qualsiasi espressione Python.

Semantica: se il **valore** di espressione è presente tra gli elementi di lista l'operatore `in` produce il valore `True`; in caso contrario produce `False`. Il comportamento dell'operatore `not in` è quello opposto. La ricerca **non** viene estesa agli elementi di eventuali liste nidificate all'interno di lista.

Esempi


Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC
v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> [7, -3, 2] == [3, 'a']
False
>>> v = [1, 2, 3]
>>> v != [4,2,5]
True
>>> -3 in [7, -3, 2]
True
>>> 5 in [7, -3, 2]
False
>>> 5 not in [7, -3, 2]
True
>>> 'a' in [7, -3, 2]
False
>>> 2 in v
True
>>>
Ln: 18 Col: 4

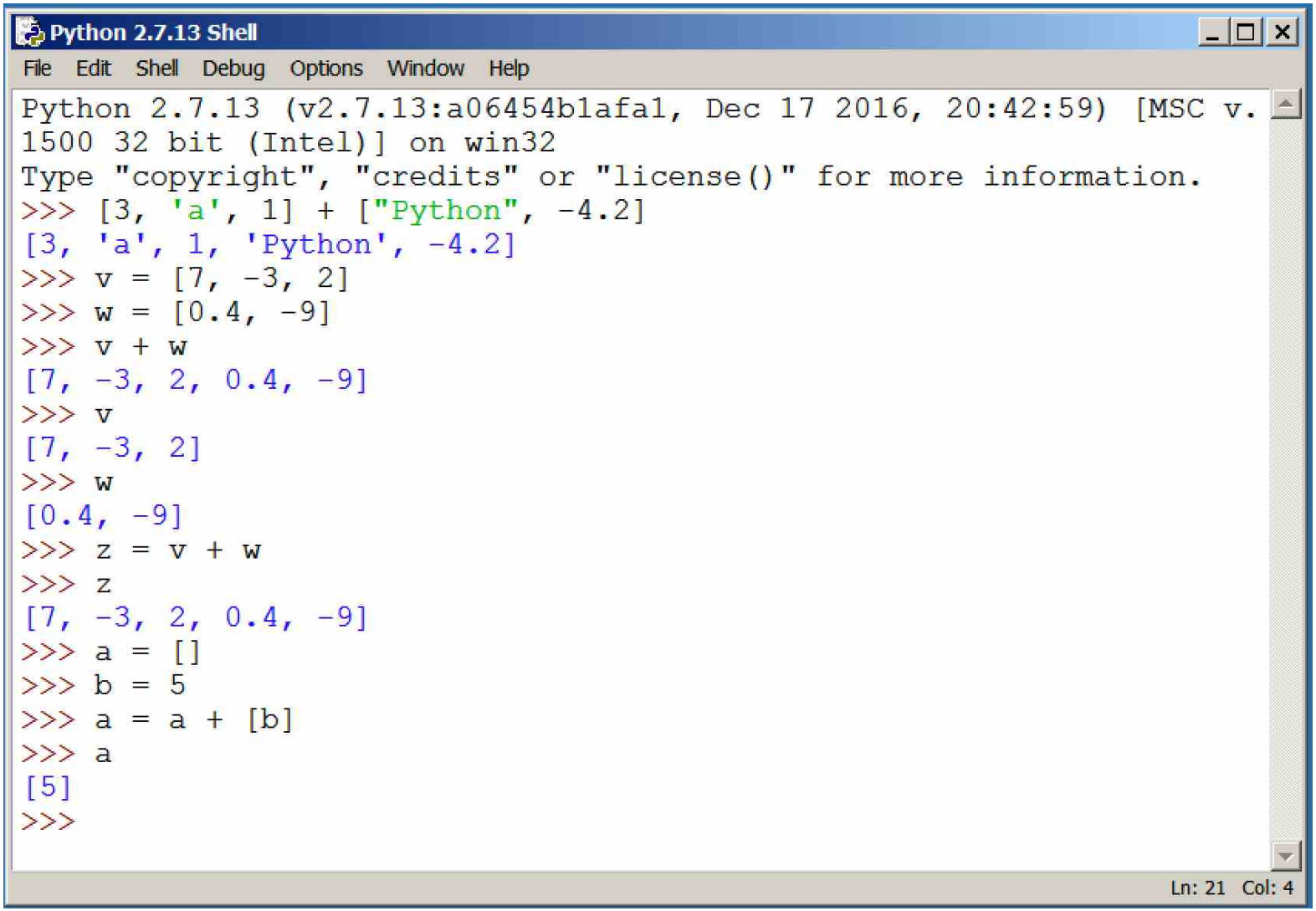
L'operatore di concatenazione

Questo operatore è analogo al corrispondente operatore del tipo di dato stringa.

Sintassi: lista_1 + lista_2

Semantica: la concatenazione produce una **nuova** lista composta dagli elementi di lista_1 seguiti da quelli di lista_2, disposti nello stesso ordine in cui si trovano nelle due liste.
Le liste originali **non** vengono modificate.

Operatori sulle liste: esempi



L'operatore di indicizzazione

L'operatore di **indicizzazione** consente di accedere a ogni singolo elemento di una lista, per mezzo dell'indice corrispondente.

Sintassi: lista[indice]
dove indice deve essere un'espressione il cui valore sia un intero compreso tra 0 e la lunghezza della lista **meno uno**.

Semantica: il risultato è il valore dell'elemento di lista nella posizione corrispondente al valore di indice. Se il valore di indice non corrisponde a una delle posizioni della lista si otterrà un messaggio d'errore.

L'operatore di indicizzazione

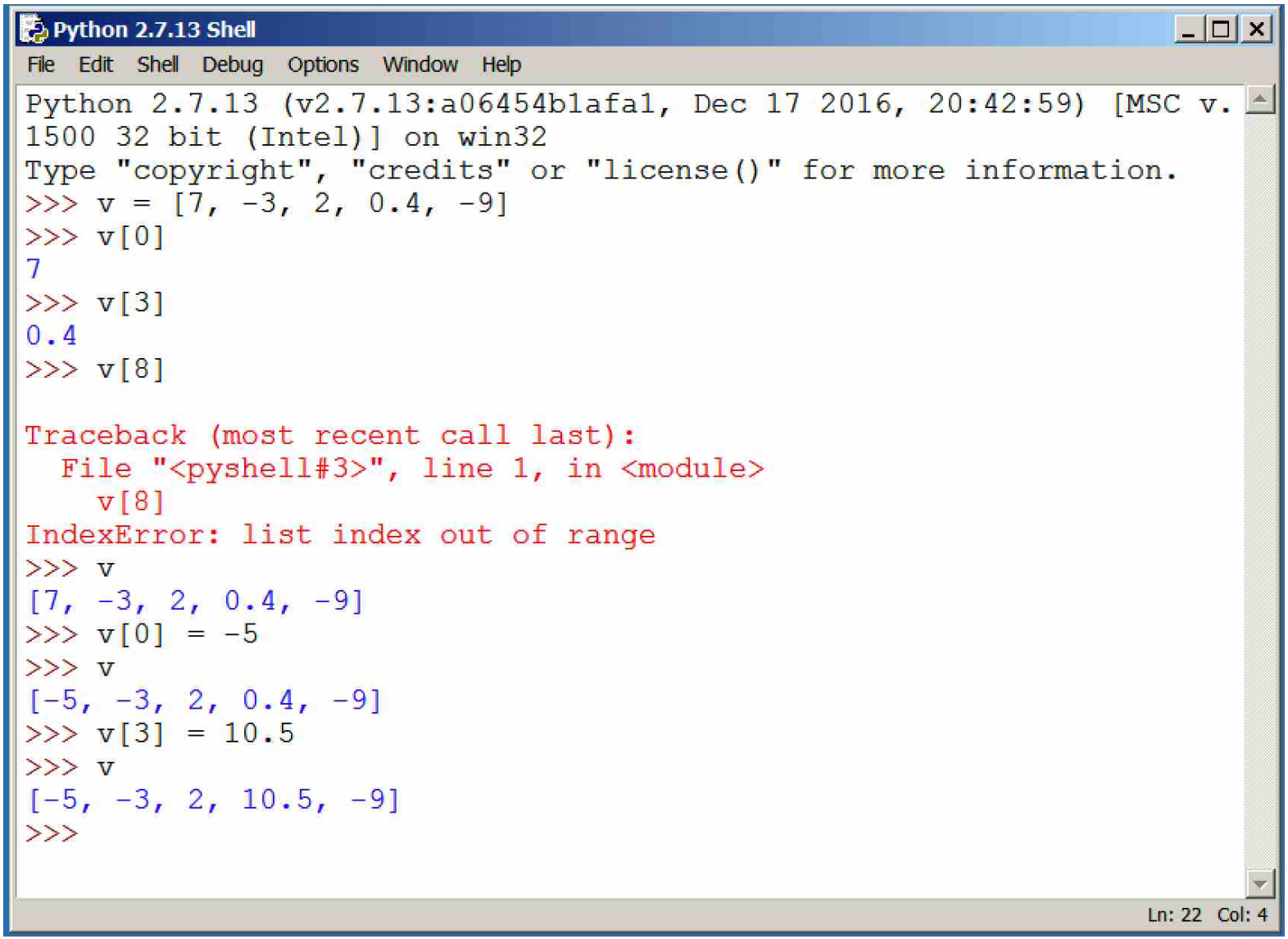
L'operatore di indicizzazione consente anche di **modificare** i singoli elementi di una lista, attraverso un'istruzione di assegnamento.

Sintassi: lista[indice] = espressione

- lista indica il nome di una variabile alla quale sia stata in precedenza assegnata una lista
- espressione indica una **qualsiasi** espressione Python

Semantica: l'elemento di lista nella posizione corrispondente a indice viene **sostituito** dal valore di espressione.

Esempi


Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.
1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> v = [7, -3, 2, 0.4, -9]
>>> v[0]
7
>>> v[3]
0.4
>>> v[8]
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
v[8]
IndexError: list index out of range
>>> v
[7, -3, 2, 0.4, -9]
>>> v[0] = -5
>>> v
[-5, -3, 2, 0.4, -9]
>>> v[3] = 10.5
>>> v
[-5, -3, 2, 10.5, -9]
>>>
Ln: 22 Col: 4

L'operatore di *slicing*

L'operatore di **slicing** ("sezionamento") restituisce una lista composta da una **sottosequenza** di una data lista.

Sintassi

- lista[indice_1:indice_2]
- lista[indice_1:]
- lista[:indice_2]
- lista[:]

dove indice_1 e indice_2 sono espressioni i cui valori devono essere numeri interi compresi tra 0 e la lunghezza di lista.

Semantica: il risultato è una lista composta dagli elementi di lista aventi indici da indice_1 a $\text{indice}_2 - 1$ (l'elemento di indice indice_2 **non** viene incluso nel risultato). Se indice_1 è omesso, viene considerato pari a 0; se indice_2 viene omesso, viene considerato pari alla lunghezza della lista. Ne consegue che lista[:] restituisce una lista identica a quella originale.

Anche in questo caso la lista originale **non** viene modificata.

L'operatore di *slicing*

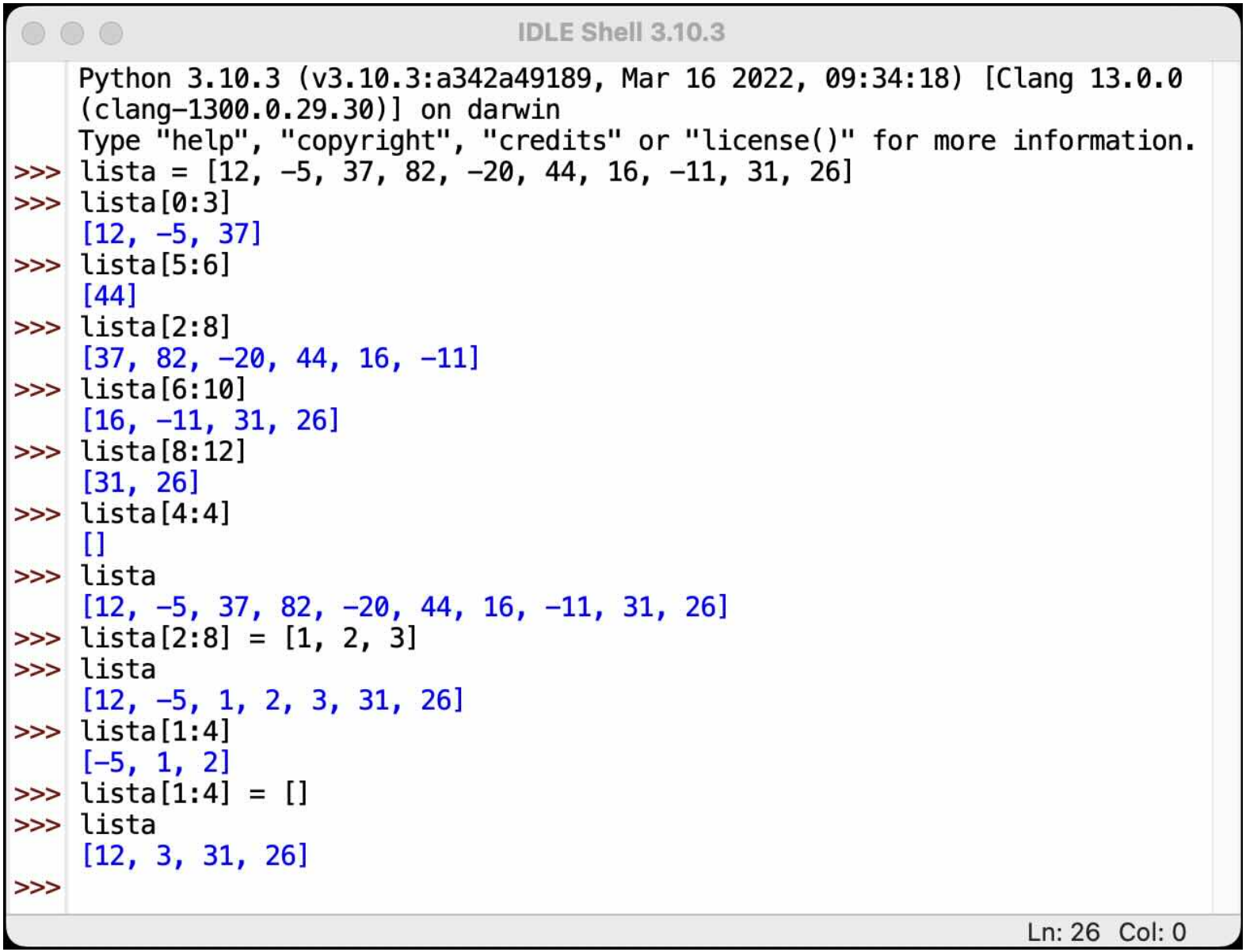
In modo analogo all'operatore di indicizzazione, anche quello di *slicing* consente di **modificare** una lista, sostituendo a una sua sottosequenza un'altra lista, di lunghezza **qualsiasi**.

Sintassi: lista`[`$\texttt{indice}_1$`:`$\texttt{indice}_2$`]` = espressione

dove espressione indica un'espressione Python che abbia per valore una lista.

Semantica: la sottosequenza di lista composta dagli elementi di lista aventi indici da indice_1 a $\text{indice}_2 - 1$ viene **sostituita** dalla lista corrispondente a espressione.

L'operatore di *slicing*: esempi

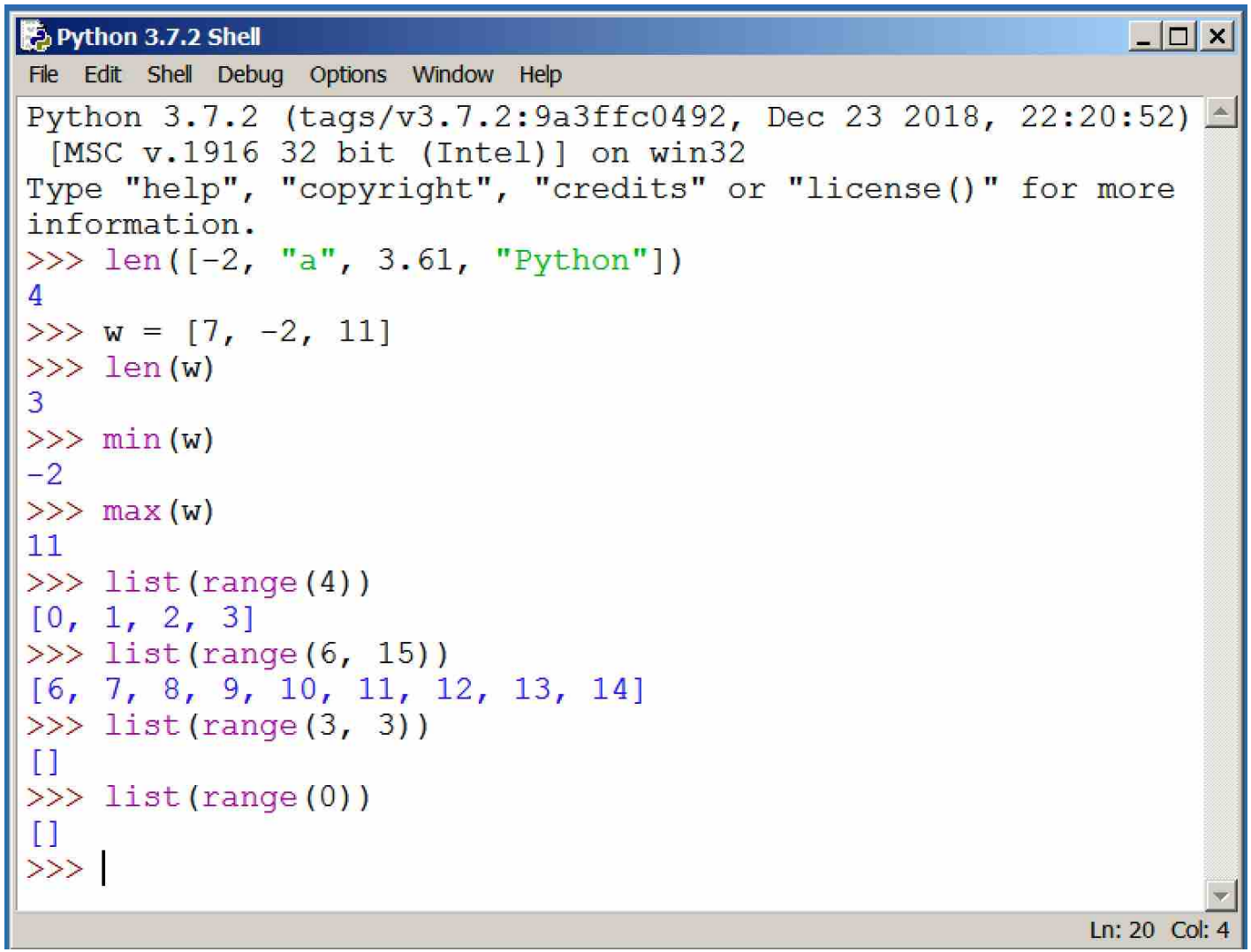


Funzioni predefinite che operano sulle liste

Alcune delle principali funzioni predefinite

nome	**descrizione**
`len(lista)`	restituisce la lunghezza di una lista
`min(lista)`	restituisce l'elemento più piccolo in una lista composta da numeri
`max(lista)`	restituisce l'elemento più grande in una lista composta da numeri
`range(a)`	restituisce un valore che può essere trasformato nella lista `[0, 1,...,a-1]` se `a` $>$ 0 (il valore associato ad `a` deve essere un intero) mediante la funzione `list`
`range(a,b)`	come sopra (i valori di `a` e `b` devono essere numeri interi): se `a` $<$ `b` consente di costruire la lista `[a,a+1,...,b-1]`
`list(x)`	costruisce una lista corrispondente al valore `x` restituito da `range`

Esempi


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>> len([-2, "a", 3.61, "Python"])
4
>>> w = [7, -2, 11]
>>> len(w)
3
>>> min(w)
-2
>>> max(w)
11
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(6, 15))
[6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range(3, 3))
[]
>>> list(range(0))
[]
>>>
Ln: 20 Col: 4

La funzione predefinita `split`

La funzione `split` è utile per l'elaborazione di stringhe e (come si vedrà più avanti) per l'elaborazione di dati acquisiti da *file* di testo.

La sintassi della chiamata è diversa da quella vista in precedenza, per motivi che esulano dagli scopi di questo corso (legati alla programmazione a oggetti):

```
stringa.split()
```

dove stringa, un'espressione avente per valore una stringa, è l'argomento della funzione.

Lo scopo di `split` è suddividere una stringa in corrispondenza dei caratteri di spaziatura (incluso il *newline*): essa restituisce una **lista** contenente le corrispondenti sottostringhe, nelle quali non vengono inclusi i caratteri di spaziatura. La stringa originale **non** viene modificata.

La funzione predefinita `split`

Come caso particolare, se una stringa contiene una sequenza di parole separate da caratteri di spaziatura, `split` consente di "separare" le singole parole (più precisamente, restituisce all'interno di una lista le stringhe corrispondenti alle singole parole, senza modificare la stringa originaria).

Un esempio: se la variabile `t` contiene la stringa:

```
"Questa è una frase."
```

la chiamata:

```
t.split()
```

restituirà la lista:

```
["Questa", "è", "una", "frase."]
```

La funzione predefinita `split`

Se si desidera suddividere una stringa in corrispondenza di una sequenza di uno o più caratteri qualsiasi, tale sequenza dovrà essere indicata (sotto forma di **una** stringa) come ulteriore argomento di `split`, con la seguente sintassi:

```
stringa.split(caratteri)
```

Per esempio, se alla variabile `t` fosse associata una stringa composta da una sequenza di numeri separati dal punto e virgola (senza spazi), come la seguente:

```
"15;1;25;9;6;21"
```

la chiamata:

```
t.split(";")
```

restituirebbe la lista:

```
["15", "1", "25", "9", "6", "21"]
```

(notare che i caratteri di separazione non vengono inclusi nel risultato).

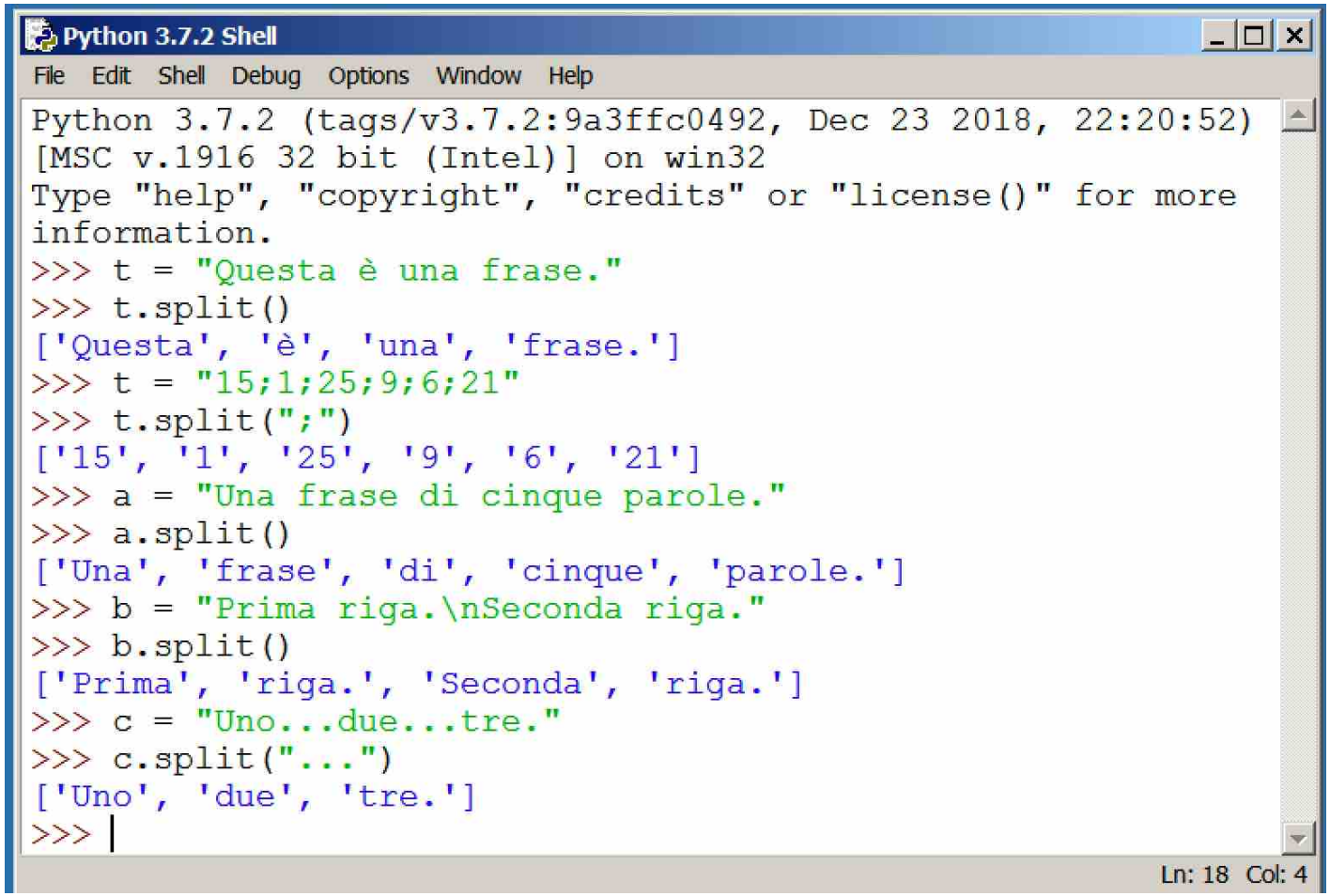
La funzione `split`: esempi

Dopo i seguenti assegnamenti:

```
a = "Una frase di cinque parole"
b = "Prima riga\nseconda riga."
c = "Uno...due...tre"
```

- `a.split()` restituisce
  `["Una", "frase", "di", "cinque", "parole"]`
- `b.split()` restituisce `["Prima riga", "seconda riga."]`
- `c.split()` restituisce `["Uno...due...tre"]`
  (la stringa non viene suddivisa, poiché non contiene caratteri di spaziatura)
- `c.split("...")` restituisce `["Uno", "due", "tre"]`

La funzione `split`: esempi



La funzione predefinita `strip`

Anche la funzione `strip` è utile per l'elaborazione di stringhe, in particolare nell'elaborazione di dati acquisiti da *file* di testo.

La sintassi della chiamata è analoga a quella di `split`:

```
stringa.strip()
```

Il valore restituito è una stringa ottenuta da quella originale eliminando gli eventuali caratteri di spaziatura (inclusi i *newline*) che si trovano all'inizio o alla fine di essa. Anche in questo caso la stringa originale **non** viene modificata.

La funzione predefinita `strip`

Se si desidera eliminare da una stringa una sequenza di uno o più caratteri qualsiasi, tale sequenza dovrà essere indicata (sotto forma di **una** stringa) come argomento di `strip`, con la seguente sintassi:

```
stringa.strip(caratteri)
```

Per esempio, se la variabile `t` fosse associata a una stringa che termina con il carattere `;` come la seguente:

```
"1527;"
```

la chiamata:

```
t.strip(";")
```

restituirebbe la stringa:

```
"1527"
```

Si noti che la stringa associata alla variabile `t` non viene modificata.

La funzione `strip`: esempi

Dopo i seguenti assegnamenti:

```
a = " Una frase di cinque parole.\n"
b = "Un esempio."
```

- `a.strip()` restituisce `"Una frase di cinque parole."`
- `b.strip()` restituisce `"Un esempio."`
  (identica alla stringa originale, poiché quest'ultima non contiene caratteri di spaziatura all'inizio né alla fine)
- `b.strip(".")` restituisce `"Un esempio"`
- `a.strip().strip(".")` restituisce
  `"Una frase di cinque parole"`
  (la prima chiamata di `strip` elimina i caratteri di spaziatura, la seconda elimina dal risultato il carattere `.` finale)

Esempio: costruzione di una lista

In molti programmi è necessario costruire una lista composta da valori che dovranno essere acquisiti **durante l'esecuzione** degli stessi programmi. La lista non può quindi essere scritta in modo esplicito al loro interno, ma dovrà essere ottenuta come risultato di un'opportuna sequenza di operazioni.

Per esempio, il seguente programma costruisce una lista di cinque numeri acquisiti tramite la tastiera, partendo da una lista vuota e usando l'operatore di concatenazione (*file*: `43_costruzione_lista.py`)

```
lista = []
print("Inserire cinque numeri.")
k = 1
while k <= 5:
    elemento = eval(input("Prossimo valore: "))
    lista = lista + [elemento]
    k = k + 1
print("La lista è:", lista)
```

Iterazione sugli elementi di una lista

Si è detto in precedenza che nell'accesso agli elementi di una lista tramite l'operatore di indicizzazione, con l'espressione lista [indice], il valore di indice può essere un'espressione qualsiasi. Questo implica che al suo interno possono comparire una o più variabili.

Per esempio, se alle variabili `k` e `v` sono stati assegnati rispettivamente i valori `3` e `[4, -3, 6, 2, 9, -12]`, l'espressione `v[k]` produrrà il valore `2`, e l'espressione `v[k+1]` produrrà il valore `9`.

L'uso di variabili per rappresentare gli indici di una lista consente di esprimere in modo conciso, attraverso un'istruzione iterativa, la ripetizione di una stessa sequenza di istruzioni su tutti gli elementi di una lista, come mostrato di seguito.

Iterazione sugli elementi di una lista

Lo schema seguente mostra come accedere in sequenza a tutti gli elementi di una lista, dal primo all'ultimo, per eseguire una stessa operazione su ciascuno di essi, mediante l'istruzione iterativa `while` e una variabile che svolge il ruolo di indice.

```
k = 0
while k < len(lista):
    istruzioni che coinvolgono lista[k]
    k = k + 1
```

Si noti l'uso della funzione `len`, che consente di applicare questo schema anche a liste delle quali non sia nota la lunghezza nel momento in cui si scrive il programma.
Notare anche la condizione `k < len(lista)` nell'istruzione `while`: il valore dell'indice dell'ultimo elemento è infatti pari alla lunghezza della lista **meno uno**.

Esempi

Il seguente programma (disponibile nel *file* `44_stampa_lista.py`) stampa tutti gli elementi di una lista acquisita attraverso la tastiera, dal primo all'ultimo

```
lista = eval(input("Inserire una lista: "))
print("Gli elementi della lista sono:")
k = 0
while k < len(lista):
    print(lista[k])
    k = k + 1
```

Esempi

Una semplice variante consente di accedere agli elementi di una lista in ordine inverso, dall'ultimo al primo.

```
sequenza = eval(input("Inserire una lista: "))
print("Gli elementi, dall'ultimo al primo, sono:")
k = len(lista) - 1
while k >= 0:
    print(lista[k])
    k = k - 1
```

Si noti che il valore iniziale della variabile `k` corrisponde all'indice dell'ultimo elemento, e che il valore associato a tale variabile viene **decrementato** di un'unità in ogni iterazione.

Esempi

Il programma riportato in basso acquisisce una lista di numeri e ne stampa i soli valori negativi
(*file*: `45_stampa_negativi_lista.py`)

```
lista = eval(input("Inserire una lista di numeri: "))
print("I valori negativi sono:")
k = 0
while k < len(lista):
    if lista[k] < 0:
        print(lista[k])
    k = k + 1
```

Esempi

La funzione seguente riceve come argomento una lista che si assume essere composta da numeri, e restituisce la loro somma (*file*: `46_somma_lista.py`). Si noti che la stessa operazione viene eseguita dalla funzione predefinita `sum`.

```
def somma_lista(lista):
    somma = 0
    k = 0
    while k < len(lista):
        somma = somma + lista[k]
        k = k + 1
    return somma
```

Esempi

Una funzione che riceve come argomento una lista che si assume essere composta da numeri, e restituisce il maggiore di tali numeri (*file*: `47_massimo_lista.py`). La stessa operazione viene eseguita dalla funzione predefinita `max`.

```
def massimo_lista(lista):
    massimo = lista[0]
    k = 1
    while k < len(lista):
        if lista[k] > massimo:
            massimo = lista[k]
        k = k + 1
    return massimo
```

Esempi

La funzione riportata di seguito verifica se gli elementi di una lista di numeri ricevuta come argomento siano ordinati in senso **non decrescente**, restituendo `True` in caso affermativo, `False` altrimenti (*file*: `48_verifica_ordinamento_lista.py`).

Una sequenza è ordinata in senso non decrescente se e solo se **ciascuno** degli elementi dal primo al **penultimo** è minore o uguale a quello **successivo**. La funzione restituisce quindi `False` non appena tale condizione risulta falsa per una coppia di elementi adiacenti.

Notare che usando la variabile `i` per memorizzare l'indice di uno degli elementi della lista, in ogni iterazione il confronto tra un coppia di elementi adiacenti può essere eseguito considerando gli indici `i` e `i + 1`. Per questo motivo il valore più grande che `i` deve assumere è pari all'indice del **penultimo** elemento della lista.

Esempi

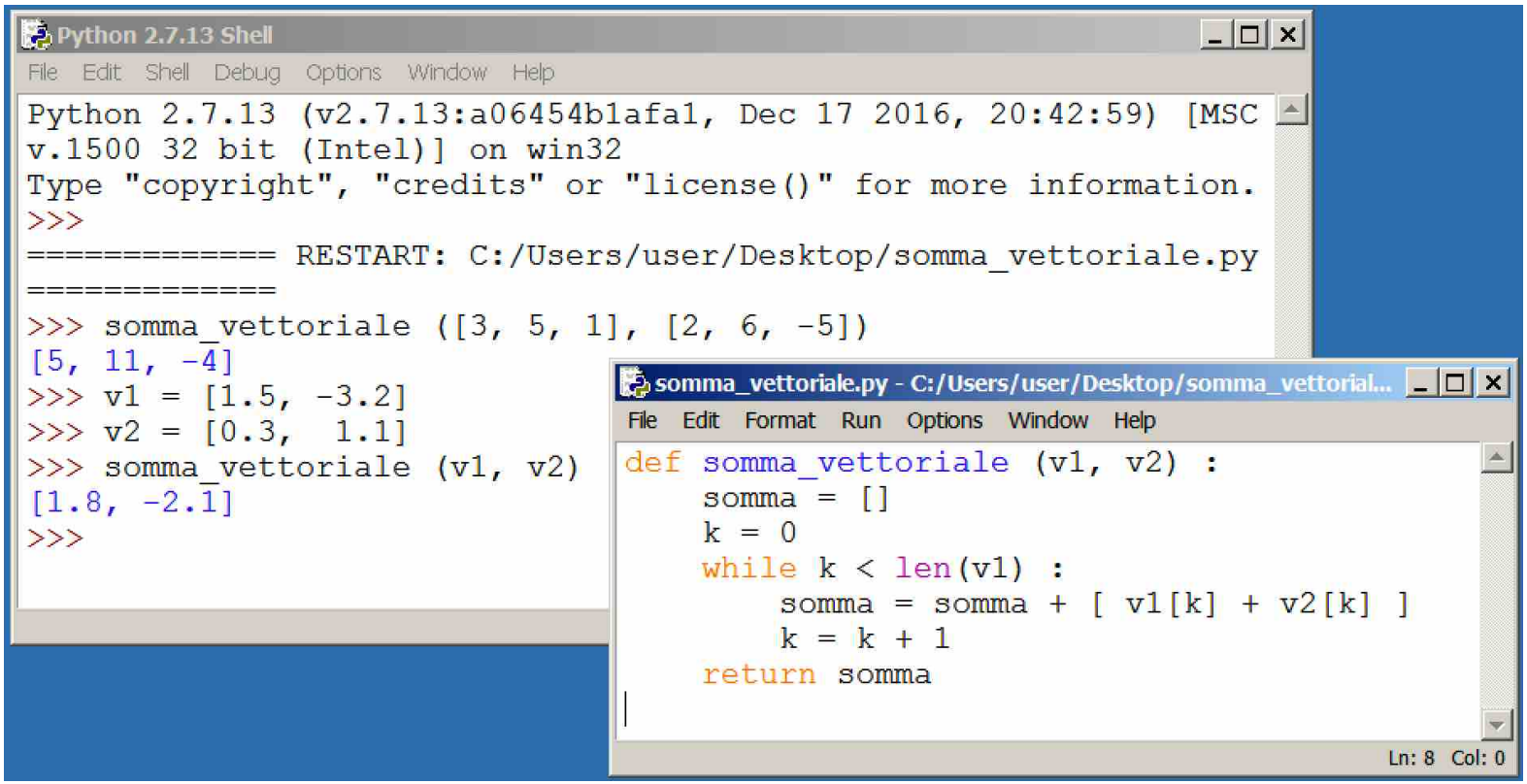
```
def ordinati(lista):
    i = 0
    while i < len(lista) - 1:
        if lista[i] > lista[i+1]:
            return False
        i = i + 1
    return True
```

Esempi

Le liste possono essere usate per memorizzare gli elementi (coordinate) di vettori. La funzione che segue (*file*: `49_somma_vettoriale.py`) esegue la somma di due vettori (liste di numeri che si assume abbiano la stessa dimensione) ricevuti come argomenti, restituendo il risultato in una lista delle stesse dimensioni. Si noti l'uso dell'operatore di concatenazione per costruire la lista che conterrà il risultato.

```
def somma_vettoriale(v1, v2):
    somma = []
    k = 0
    while k < len(v1):
        somma = somma + [v1[k] + v2[k]]
        k = k + 1
    return somma
```

Esempio


Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC
v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/user/Desktop/somma_vettoriale.py
==========
>>> somma_vettoriale ([3, 5, 1], [2, 6, -5])
[5, 11, -4]
>>> v1 = [1.5, -3.2]
>>> v2 = [0.3, 1.1]
>>> somma_vettoriale (v1, v2)
[1.8, -2.1]
>>>
somma_vettoriale.py - C:/Users/user/Desktop/somma_vettorial...
File Edit Format Run Options Window Help
def somma_vettoriale (v1, v2) :
somma = []
k = 0
while k < len(v1) :
somma = somma + [v1[k] + v2[k]]
k = k + 1
return somma
Ln: 8 Col: 0

Stringhe e liste: *sequenze*

Le stringhe e le liste sono tipi di dato che hanno in comune il fatto di essere costituite da **sequenze ordinate** di un numero qualsiasi di elementi. Per questo motivo sono entrambe indicate in linguaggio Python con il termine più generale di **sequenze**.

Alcuni operatori e alcune funzioni predefinite che operano su **sequenze ordinate** possono essere applicati sia alle **liste** che alle **stringhe** (come si è già visto per l'operatore di concatenazione)

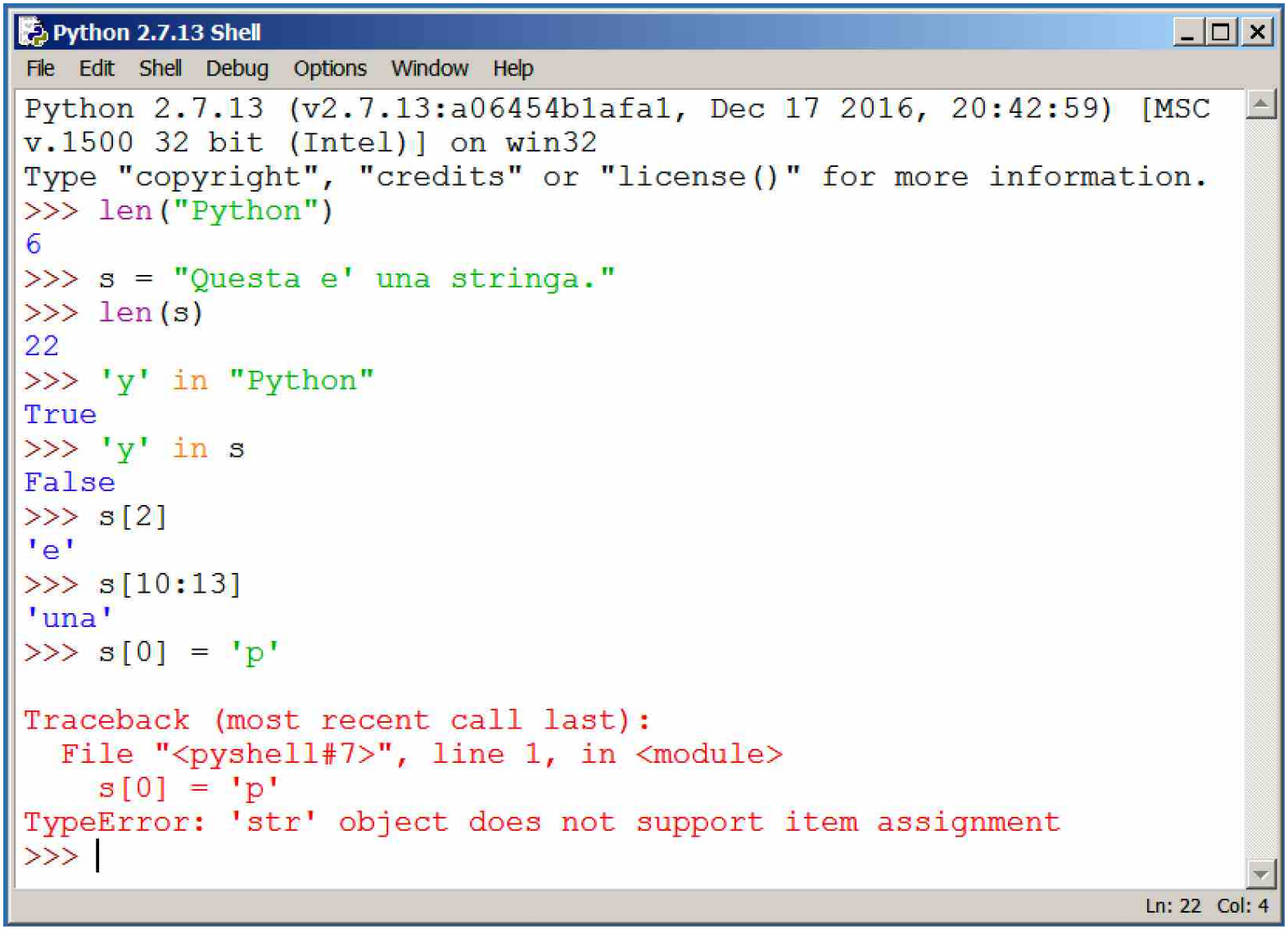
- funzioni predefinite: `len`
- operatori: `in` e `not in`, indicizzazione, *slicing*

Stringhe e liste: *sequenze*

Liste e stringhe presentano però una differenza fondamentale

- i valori di tipo *lista* sono **mutabili**: attraverso l'istruzione di assegnamento e l'operatore di indicizzazione si è visto che è possibile **modificare** i singoli elementi di una lista
- i valori di tipo *lista* **non** sono mutabili (il tentativo di modificare un elemento di una stringa produce un errore)

Esempi


Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC
v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> len("Python")
6
>>> s = "Questa e' una stringa."
>>> len(s)
22
>>> 'y' in "Python"
True
>>> 'y' in s
False
>>> s[2]
'e'
>>> s[10:13]
'una'
>>> s[0] = 'p'
Traceback (most recent call last):
File "<pyshell#7>", line 1, in <module>
s[0] = 'p'
TypeError: 'str' object does not support item assignment
>>>
Ln: 22 Col: 4

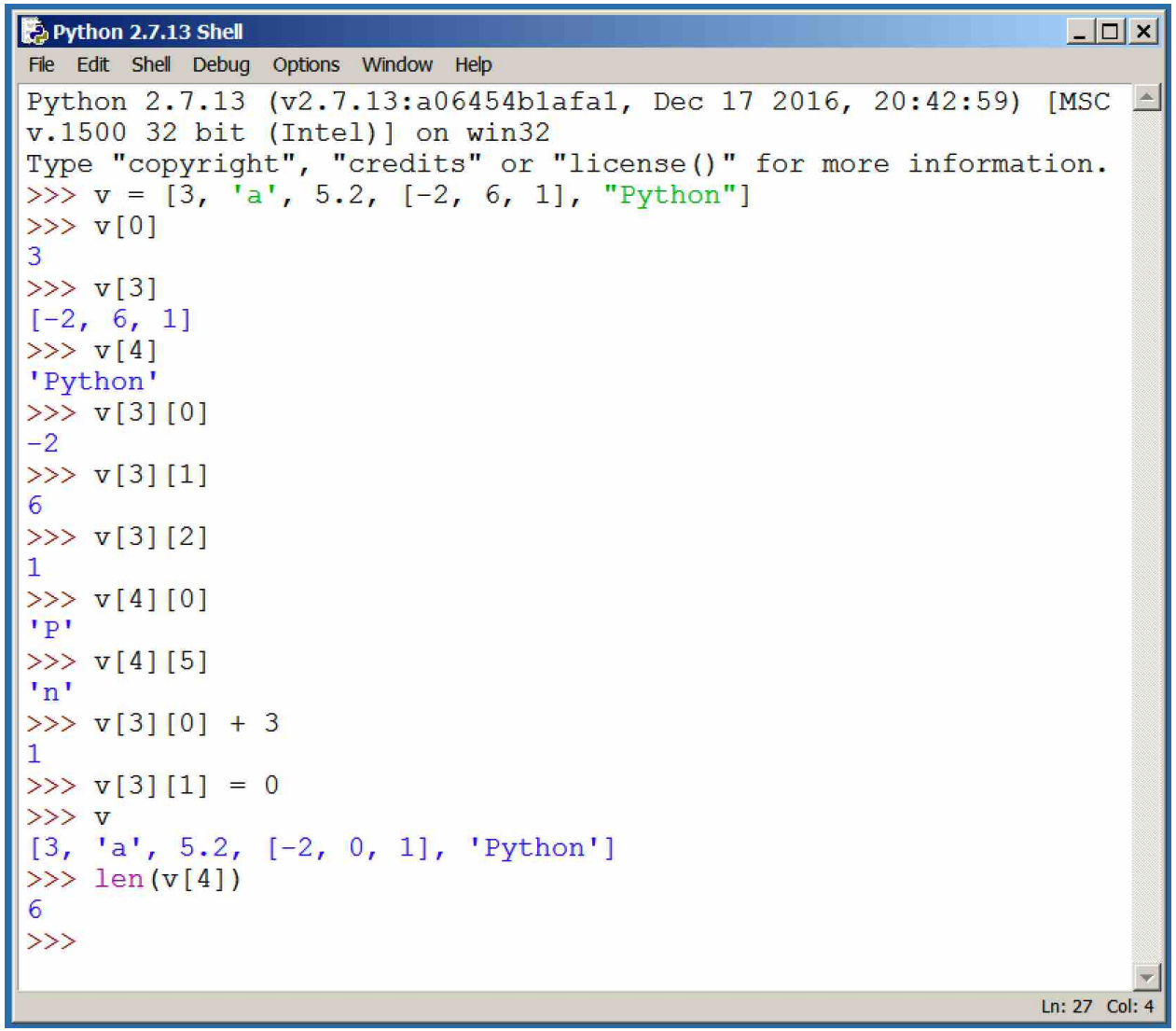
Accesso agli elementi di sequenze nidificate

Si è detto che gli elementi di una lista possono essere valori di tipi qualsiasi, quindi anche strutturati, come stringhe o altre liste.

L'operatore di indicizzazione consente di accedere anche agli elementi di strutture **nidificate**.

Se s è una variabile a cui è stata assegnata una lista, e l'elemento di indice *i* è a sua volta una sequenza (lista o stringa), sarà possibile accedere all'elemento di indice *j* di quest'ultima con la seguente **sintassi**: `s[i][j]`

Esempi


Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> v = [3, 'a', 5.2, [-2, 6, 1], "Python"]
>>> v[0]
3
>>> v[3]
[-2, 6, 1]
>>> v[4]
'Python'
>>> v[3][0]
-2
>>> v[3][1]
6
>>> v[3][2]
1
>>> v[4][0]
'P'
>>> v[4][5]
'n'
>>> v[3][0] + 3
1
>>> v[3][1] = 0
>>> v
[3, 'a', 5.2, [-2, 0, 1], 'Python']
>>> len(v[4])
6
>>>
Ln: 27 Col: 4

Liste nidificate: esempio

Si è visto che una delle possibili applicazioni delle liste è la rappresentazione di vettori. Analogamente, le liste nidificate possono essere usate per rappresentare **matrici**, cioè entità bidimensionali.

Una matrice di m righe e n colonne può essere rappresentata in un programma Python in diversi modi.

Una possibilità consiste nel memorizzare i suoi elementi, in un ordine opportuno, in una lista "semplice" (non nidificata) di $m \times n$ elementi.

Si consideri per esempio la seguente matrice:

$$\begin{pmatrix} -3 & 1 & 4 \\ 2 & 5 & -1 \end{pmatrix}$$

Procedendo dall'alto verso il basso e da destra verso sinistra, tale matrice può essere rappresentata dalla lista `[-3, 1, 4, 2, 5, -1]`.

Liste nidificate: esempio

Data una matrice di m righe e n colonne, non è difficile vedere che la scelta precedente associa l'elemento nella i-esima riga e nella j-esima colonna all'elemento di una lista nella posizione $(i-1)n+j$, avente quindi indice $(i-1)n+j-1$.

Per esempio, se la matrice mostrata in precedenza ($m=2$ righe e $n=3$ colonne) fosse memorizzata in una lista assegnata a una variabile di nome `M`, l'elemento nella seconda riga ($i=2$) e nella prima colonna ($j=1$) corrisponderebbe all'elemento nella quarta posizione della lista: $(i-1)n+j=4$. Per accedere a tale elemento si dovrebbe quindi usare l'espressione `M[3]`.

Liste nidificate: esempio

È però possibile rappresentare una matrice con una lista nidificata, in modo da ottenere una corrispondenza più diretta e intuitiva tra gli indici di riga e colonna di un elemento della matrice e quelli dello stesso elemento della lista.

Questo risultato si ottiene considerando una matrice come una **sequenza ordinata** di m righe, ciascuna delle quali è a sua volta una **sequenza ordinata** di n valori semplici (numeri reali).
Ogni riga può quindi essere rappresentata come una lista di n numeri, mentre l'intera matrice sarà rappresentata da una lista di m liste (righe della matrice).

In questo modo l'elemento della matrice nella i-esima riga e nella j-esima colonna corrisponderà all'elemento della lista avente indici $i-1$ e $j-1$.

Liste nidificate: esempio

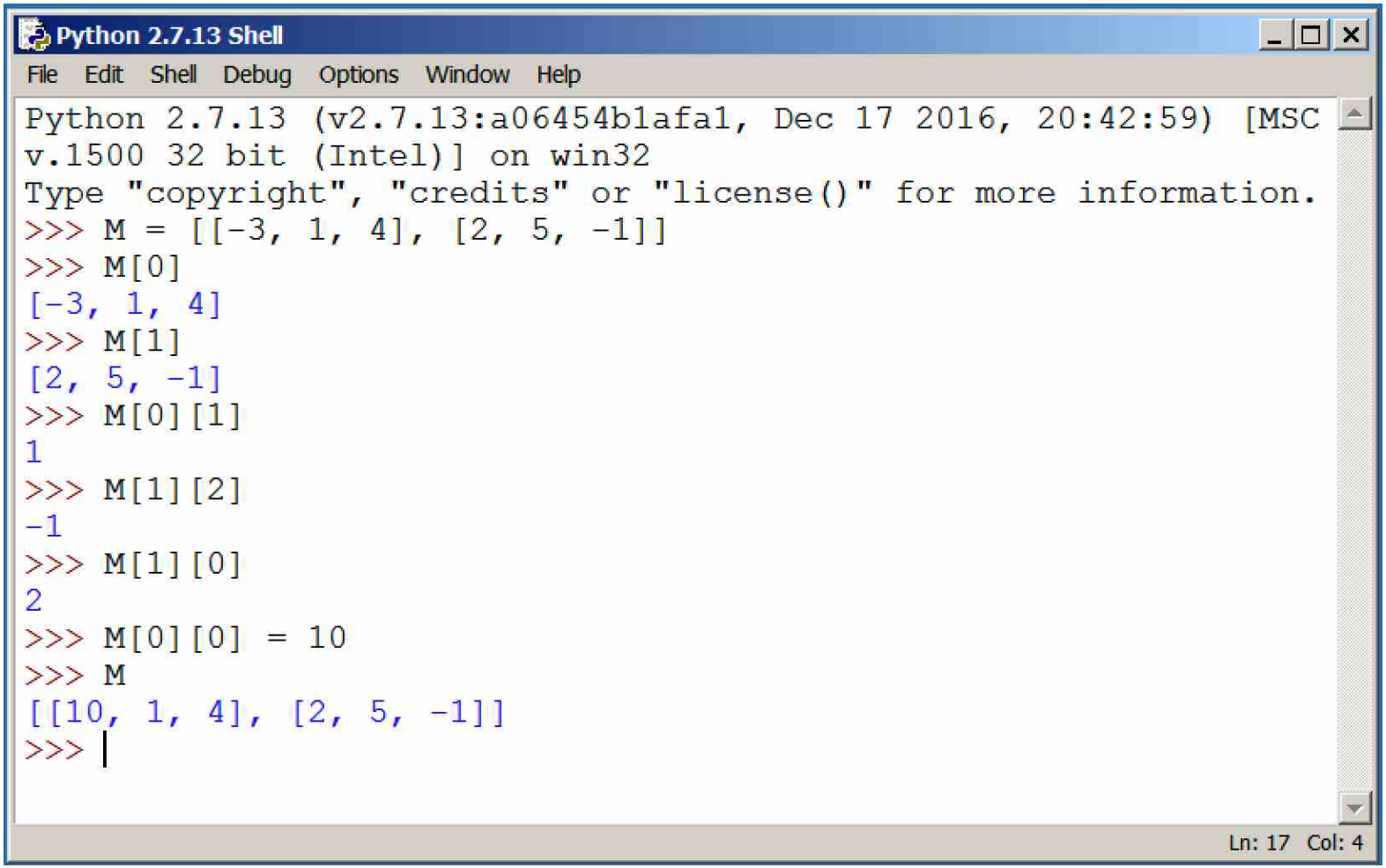
La matrice considerata in precedenza può essere rappresentata dalla seguente lista nidificata:

```
[[-3, 1, 4], [2, 5, -1]]
```

Assumendo come in precedenza che tale lista sia stata assegnata a una variabile di nome `M`, per accedere all'elemento nella seconda riga ($i = 2$) e nella prima colonna ($j = 1$) si userà l'espressione:

```
M[1][0]
```

Liste nidificate: esempio



Liste nidificate: esempio

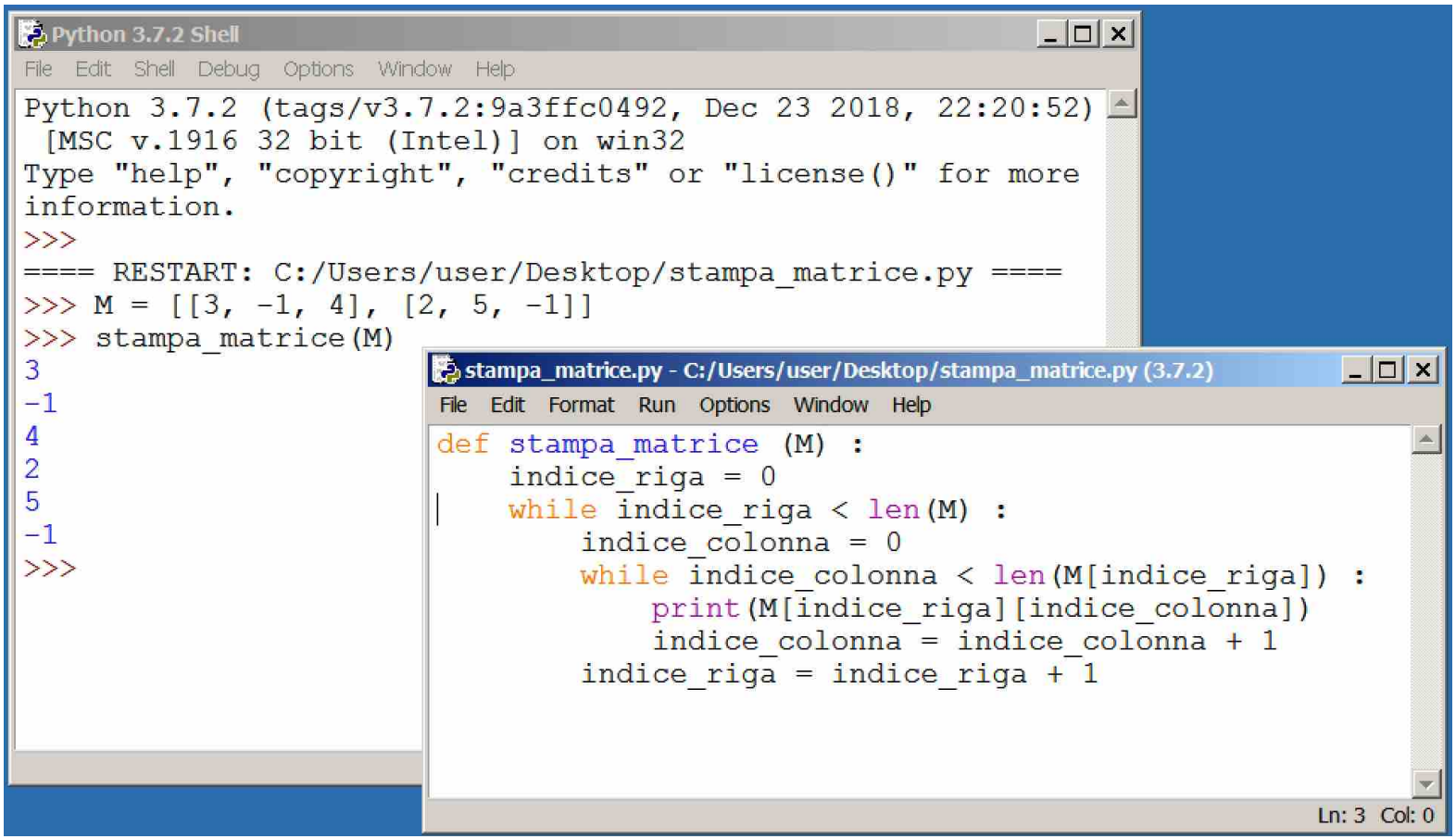
Data una matrice memorizzata in una lista nidificata, per accedere ai suoi elementi secondo un certo ordine (per es., dalla riga in alto a quella in basso, e dalla colonna a sinistra a quella a destra) si dovranno usare due istruzioni iterative nidificate: la prima farà variare l'indice di riga, la seconda l'indice di colonna. Tali indici dovranno ovviamente essere memorizzati in due variabili distinte.

La funzione mostrata di seguito stampa gli elementi di una matrice nell'ordine indicato sopra (*file*: `50_stampa_matrice.py`).

Liste nidificate: esempio

```
def stampa_matrice(M):
    indice_riga = 0
    while indice_riga < len(M):
        indice_colonna = 0
        while indice_colonna < len(M[indice_riga]):
            print(M[indice_riga][indice_colonna])
            indice_colonna = indice_colonna + 1
        indice_riga = indice_riga + 1
```

Liste nidificate: esempio

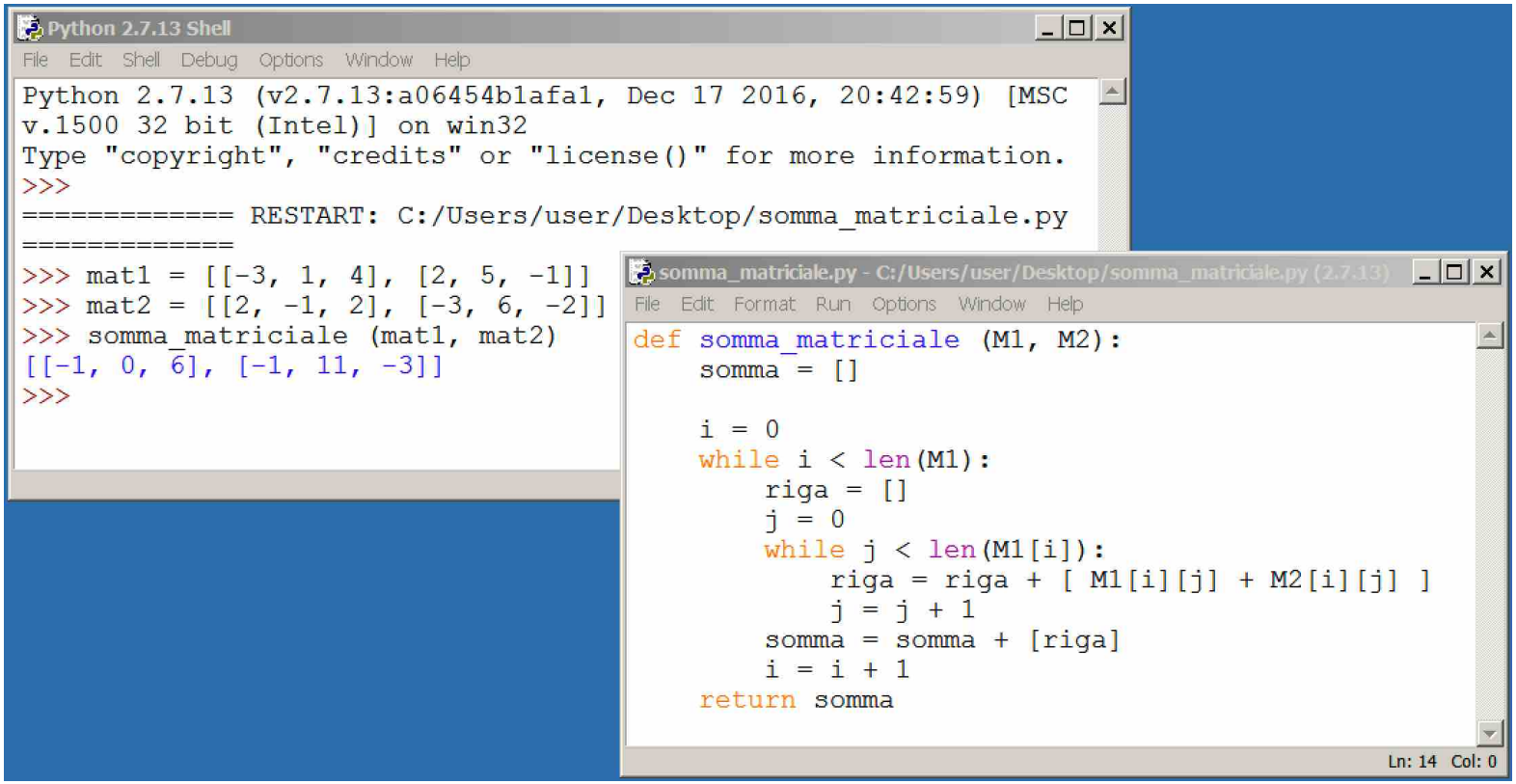


Liste nidificate: esempio

La funzione seguente (*file*: `51-somma_matriciale.py`) restituisce la somma di due matrici (che si assumono avere le stesse dimensioni) ricevute come argomento. Come nel caso della somma vettoriale, la lista contenente il risultato viene costruita per concatenazione.

```
def somma_matriciale(M1, M2):
    somma = []
    i = 0
    while i < len(M1):
        riga = []
        j = 0
        while j < len(M1[i]):
            riga = riga + [M1[i][j] + M2[i][j]]
            j = j + 1
        somma = somma + [riga]
        i = i + 1
    return somma
```

Liste nidificate: esempio



L'istruzione iterativa `for`

Come altri linguaggi, anche Python include una versione alternativa dell'istruzione iterativa `while`: l'istruzione `for`.

Nel caso di Python l'istruzione `for` consente di esprimere un solo tipo di iterazione che consiste nell'accedere a **tutti** gli elementi di una **sequenza** (stringa o lista); l'accesso avviene dal primo all'ultimo elemento, e non è possibile modificare tale ordine.

In questo tipo di operazione i vantaggi dell'istruzione `for` rispetto a `while` consistono in una maggiore concisione e nella possibilità di non usare indici espliciti, come si vedrà tra poco.

L'istruzione iterativa `for`: sintassi

```
for v in s:
    sequenza di istruzioni
```

- v deve essere il nome di una variabile che non sia già usata per memorizzare dati all'interno dello stesso programma
- s deve essere un'espressione avente come valore una sequenza (una lista o una stringa)
- sequenza di istruzioni è una sequenza di una o più istruzioni qualsiasi che devono essere scritte rispettando la stessa regola sui rientri già vista per l'istruzione `while`, e che di norma eseguono un'operazione sulla variabile v

L'istruzione iterativa `for`: semantica

La sequenza di istruzioni viene eseguita per un numero di volte pari alla lunghezza di s.

Nella *i*-esima iterazione, prima dell'esecuzione di sequenza di istruzioni viene assegnato alla variabile v il valore dell'*i*-esimo elemento di s; la sequenza di istruzioni può quindi elaborare tale valore accedendo a v.

Si noti che l'istruzione `for` non richiede l'uso esplicito degli indici per accedere agli elementi di una sequenza.

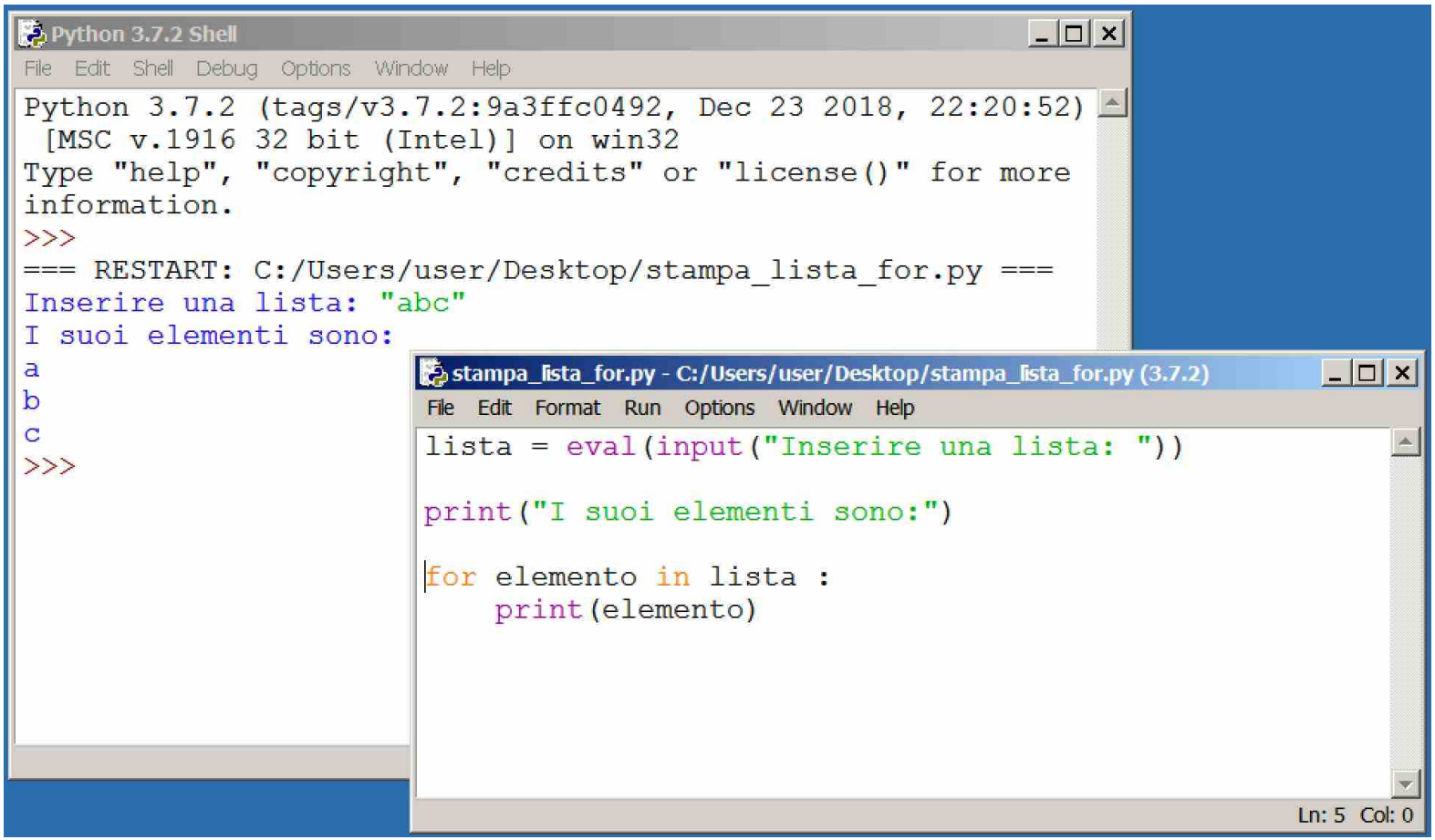
Di seguito si mostra l'uso dell'istruzione `for` con gli stessi esempi visti in precedenza per l'accesso a sequenze mediante `while`.

Esempi

Un programma che stampa gli elementi di una lista acquisita attraverso la tastiera (*file*: `52_stampa_sequenza.py`)

```
lista = eval(input ("Inserire una lista: "))
print("I suoi elementi sono:")
for elemento in lista:
    print(elemento)
```

Esempi

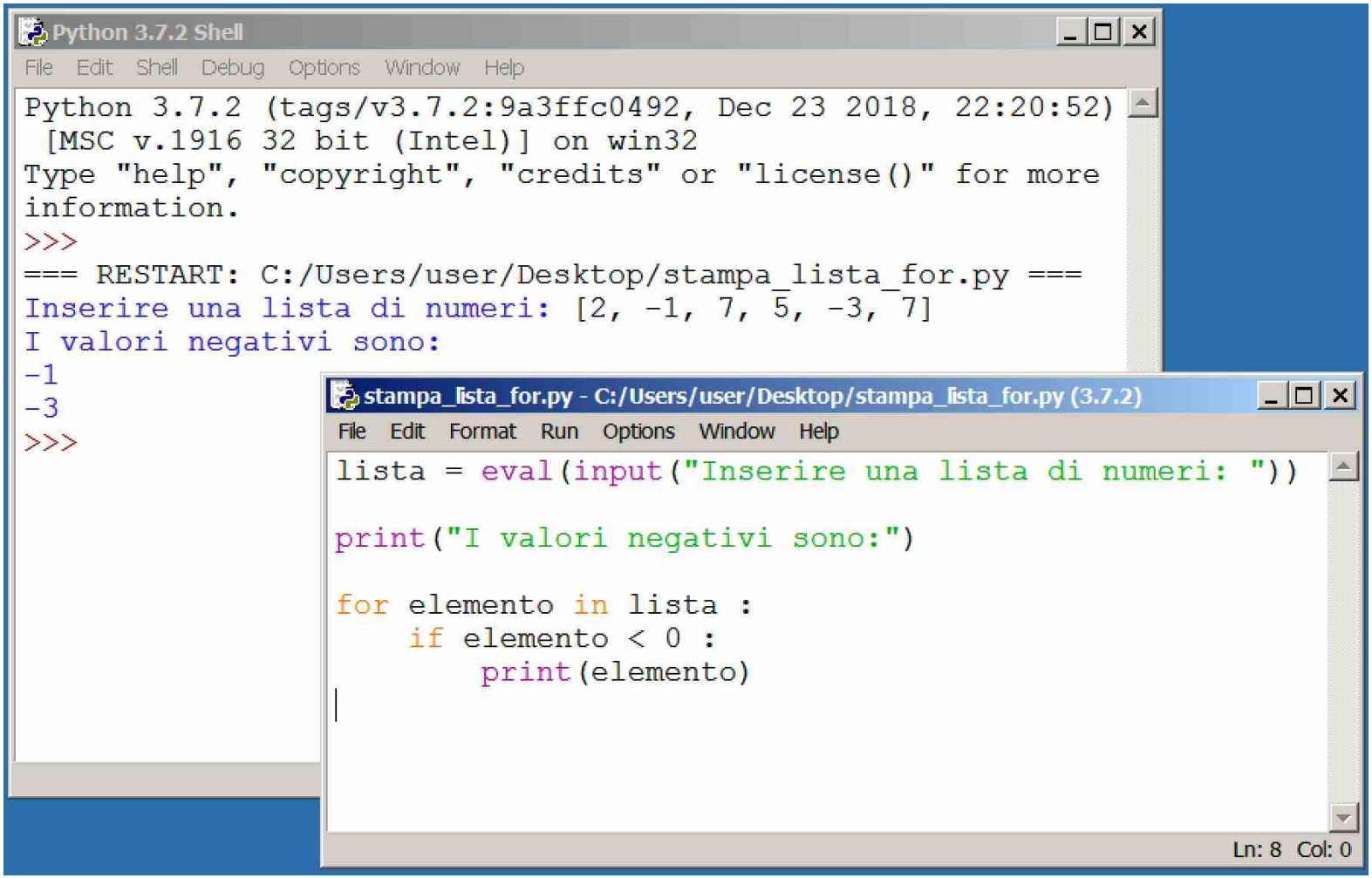

Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:/Users/user/Desktop/stampa_lista_for.py ===
Inserire una lista: "abc"
I suoi elementi sono:
a
b
c
>>>
stampa_lista_for.py - C:/Users/user/Desktop/stampa_lista_for.py (3.7.2)
File Edit Format Run Options Window Help
lista = eval(input("Inserire una lista: "))
print("I suoi elementi sono:")
for elemento in lista :
print(elemento)
Ln: 5 Col: 0

Esempi

Un programma che stampa i numeri negativi contenuti in una lista acquisita attraverso la tastiera. Il programma è disponibile nel *file* `53_stampa_negativi_lista_for.py`

```
lista = eval(input("Inserire una lista di numeri: "))
print("I valori negativi sono:")
for elemento in lista:
    if elemento < 0:
        print(elemento)
```

Esempi


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>>
=== RESTART: C:/Users/user/Desktop/stampa_lista_for.py ===
Inserire una lista di numeri: [2, -1, 7, 5, -3, 7]
I valori negativi sono:
-1
-3
>>>
stampa_lista_for.py - C:/Users/user/Desktop/stampa_lista_for.py (3.7.2)
File Edit Format Run Options Window Help
lista = eval(input("Inserire una lista di numeri: "))
print("I valori negativi sono:")
for elemento in lista :
if elemento < 0 :
print(elemento)
Ln: 8 Col: 0

Esempi

Una funzione che restituisce la somma degli elementi (che si assumono essere numeri) di una lista ricevuta come argomento (*file*: `54_somma_lista_for.py`)

```
def somma_lista(lista):
    somma = 0
    for numero in lista:
        somma = somma + numero
    return somma
```

Esempi

Una funzione che riceve come argomento una lista che si assume essere composta da numeri, e restituisce il maggiore di essi (*file*: `55_massimo_lista_for.py`)

```
def massimo_lista(lista):
    massimo = lista[0]
    for numero in lista:
        if numero > massimo:
            massimo = numero
    return massimo
```

Esempi

Una funzione che stampa tutti gli elementi di una matrice memorizzata (per righe) in una lista nidificata ricevuta come argomento (*file*: `56_stampa_matrice_for.py`)

```
def stampa_matrice(M):
    for riga in M:
        for elemento in riga:
            print(elemento)
```

Confronto tra `while` e `for`

Si consideri un'iterazione realizzata con l'istruzione `while` secondo lo schema seguente, nel quale k indica una variabile usata come indice di una sequenza s:

```
k = 0
while k < len(s):
    istruzioni che elaborano il valore s[k]
    k = k + 1
```

Non è difficile vedere che la stessa operazione può essere realizzata usando l'istruzione `for`, come segue:

```
for v in s:
    istruzioni che elaborano v
```

In questo caso l'uso di `for` è preferibile all'uso di `while` poiché consente una maggiore concisione, ed evita al programmatore di dover gestire una variabile con il ruolo di indice.

Confronto tra `while` e `for`

Si osservi tuttavia che non è possibile usare `for` per accedere a una sequenza in un ordine che non sia dal primo all'ultimo elemento, né per eseguire operazioni che richiedono l'uso esplicito degli indici, come negli esempi seguenti

- **modificare** il valore di un elemento di una lista, attraverso un'istruzione di assegnamento del tipo:
  lista[k] = valore
- accedere nello stesso passo di un'iterazione a **più di un elemento** di una sequenza, per esempio per confrontare i valori di due elementi adiacenti:

```
if s[k] != s[k + 1]:
    ...
```

Confronto tra `while` e `for`

Come esempio si considerino le istruzioni seguenti, che assegnano il valore 0 a tutti gli elementi di una lista precedentemente memorizzata in una variabile di nome `lista`:

```
k = 0
while k < len(lista):
    lista[k] = 0
    k = k + 1
```

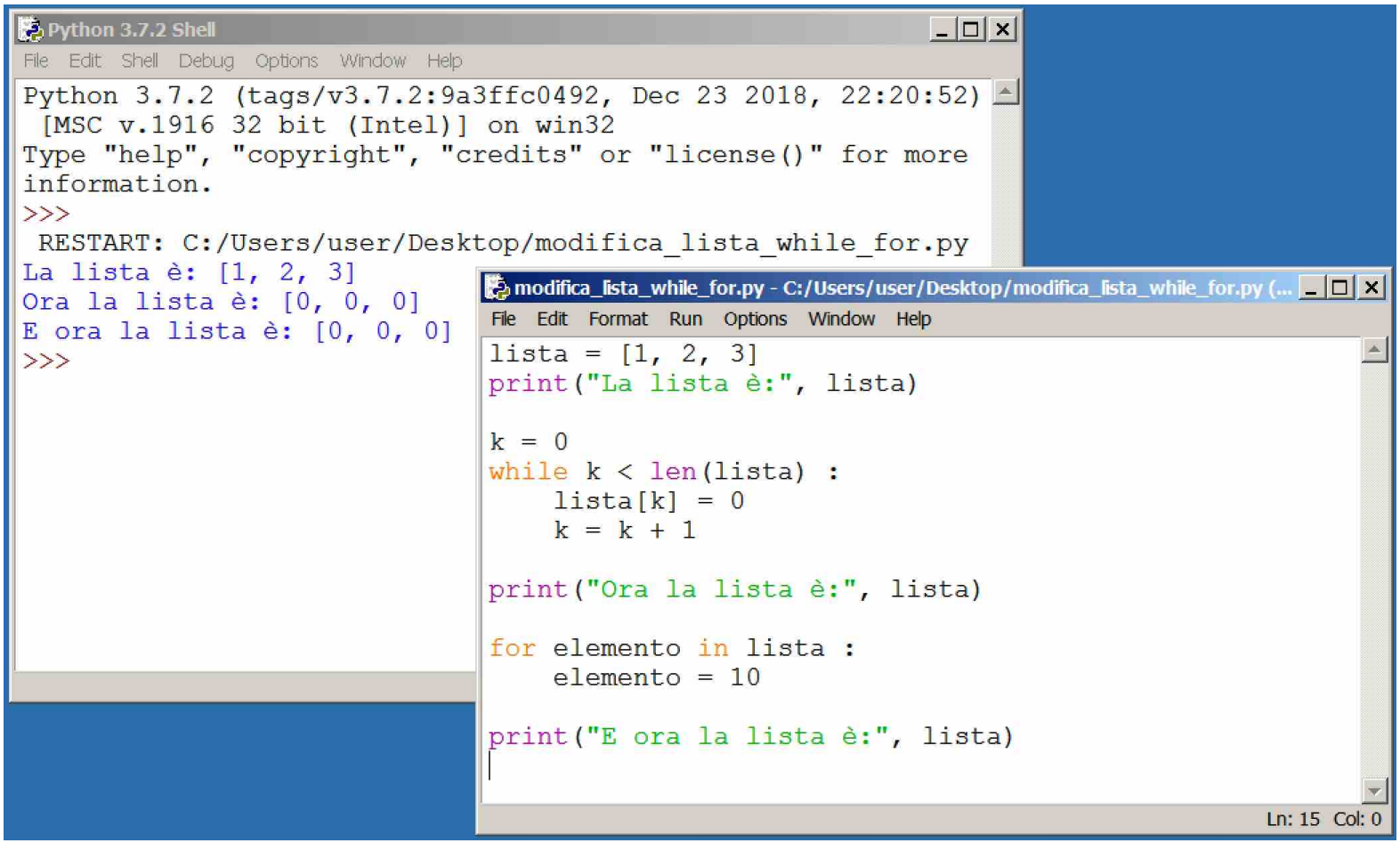
Se si cercasse di eseguire la stessa operazione mediante `for`:

```
for elemento in lista:
    elemento = 0
```

cambierebbe il valore associato alla variabile `elemento`, ma **non** il contenuto della lista.

Esempio

Tra le due iterazioni di questo programma, solo quella realizzata con `while` consente di modificare i valori degli elementi di una lista:



Confronto tra `while` e `for`

È comunque possibile, anche con l'istruzione `for`, accedere agli elementi di una sequenza `s` usando una variabile come indice: a questo scopo l'istruzione `for` deve essere applicata non a `s`, ma a una lista `L` che contenga i valori degli indici di `s`, nell'ordine desiderato, secondo lo schema seguente:

```
for k in L:
    istruzioni che accedono a s[k]
```

In particolare, se si vuole accedere a `s` dal primo all'ultimo elemento, la lista `L` dovrà contenere gli interi da 0 alla lunghezza di `s` meno uno, e quindi può essere ottenuta usando la funzione predefinita `range` già vista in precedenza:

```
for k in range(len(s)):
    istruzioni che accedono a s[k]
```

Esempio

Una seconda versione della funzione che verifica se una lista di numeri sia ordinata in senso non decrescente, usando l'istruzione `for` (*file*: `57_verifica_ordinamento_lista_for.py`)

```
def ordinati(lista):
    for i in range(len(lista) - 1):
        if lista[i] > lista[i + 1]:
            return False
    return True
```

Liste e istruzione di assegnamento

Si consideri un'istruzione di assegnamento come la seguente, assumendo che alla variabile `x` sia già stato assegnato un valore **qualsiasi**:

```
y = x
```

Tale istruzione fa sì che alla variabile `y` venga associato lo **stesso** valore associato a `x`.

Se alla variabile `x` viene successivamente associato **un altro** valore con un'istruzione di assegnamento, il valore associato a `y` **non cambia**.

Liste e istruzione di assegnamento: esempio

Dopo la seguente sequenza di istruzioni:

```
a = -3.2
b = "Python"
c = [0, 1, 2, 3]
p = a
q = b
r = c
a = "Alice"
b = [1, "Bob", 4]
c = 4
```

- il valore associato alle variabili `p`, `q` e `r` sarà, rispettivamente, `-3.2`, `"Python"` e `[0, 1, 2, 3]`
- il valore associato alle variabili `a`, `b` e `c` sarà, rispettivamente, `"Alice"`, `[1, "Bob", 4]` e `4`

Liste e istruzione di assegnamento

Si consideri ora il caso in cui il valore associato alla variabile `x` sia una **lista**.

Dato che l'istruzione `y = x` associa a `y` lo stesso valore associato a `x`, l'uso dell'operatore di **indicizzazione** su `x`, oppure su `y`, per modificare un elemento della lista mediante un'istruzione di assegnamento, si riflette sul valore associato all'altra variabile.

Liste e istruzione di assegnamento

Per esempio, dopo la seguente sequenza di istruzioni:

```
x = [1, 2, 3]
y = x
```

il valore associato a `x` e a `y` è `[1, 2, 3]` (come si può verificare tramite la *shell* dell'ambiente `IDLE`, stampando sullo schermo i valori delle due variabili).

Se successivamente si esegue l'assegnamento:

```
x[0] = 10
```

il valore associato a **entrambe** le variabili diventa `[10, 2, 3]`.

Liste come argomenti di funzioni

Per lo stesso motivo, se si passasse una lista come argomento di una funzione, ogni modifica eseguita sulla lista associata al parametro della funzione si rifletterebbe anche sulla lista del programma chiamante.

Si consideri per esempio la seguente funzione:

```
def azzera(v):
    v = 0
```

Se, dopo la definizione della funzione `azzera`, si eseguono nella *shell* le seguenti istruzioni:

```
w = 1
azzera(w)
print(w)
```

si vedrà che il valore associato alla variabile `w` è ancora `1`.

Liste come argomenti di funzioni

Si consideri invece la seguente funzione, nella quale si assume che il valore dell'argomento sia una lista:

```
def azzera(v):
    i = 0
    while i < len(v):
        v[i] = 0
        i = i + 1
```

Se, dopo la definizione di tale funzione, si eseguono nella *shell* le seguenti istruzioni:

```
w = [1, 2, 3]
azzera(w)
print(w)
```

si vedrà che il valore associato a `w` è diventato `[0, 0, 0]`.

Uso dell'operatore di *slicing* per la copia di una lista

Se il valore associato a una variabile (per es., `x`) fosse una lista, e si volesse associare a un'altra variabile (per es., `y`) una **copia** della stessa lista, si potrebbe usare l'operatore di *slicing*:

```
y = x[:]
```

Esempi

- dopo le seguenti istruzioni:

  ```
  x = [1, 2, 3]
  y = x[:]
  x[0] = 10
  ```

  il valore associato a `x` è `[10, 2, 3]`, quello associato a `y` è ancora `[1, 2, 3]`

- analogamente, dopo le istruzioni:

  ```
  w = [1, 2, 3]
  azzera(w[:])
  ```

  il valore associato a `w` è ancora `[1, 2, 3]`

Il tipo di dato *dizionario*

Le liste consentono di rappresentare dati strutturati i cui valori siano composti da una **sequenza ordinata** di valori più semplici.

Un altro caso comune nella pratica è quello in cui i valori dei dati da elaborare siano rappresentabili come **collezioni** (insiemi) di valori più semplici e **non ordinati**, ciascuno dei quali abbia un significato che possa essere descritto con un **nome simbolico**.

Un esempio sono le informazioni anagrafiche su una persona: nome, cognome, data e luogo di nascita, codice fiscale, ecc.

I dati aventi tali caratteristiche possono essere rappresentati in Python per mezzo del tipo strutturato **dizionario**.

Il tipo di dato *dizionario*

I valori del tipo di dato **dizionario** sono collezioni non ordinate di un numero qualsiasi di elementi, ciascuno dei quali è composto da un valore di un tipo **qualsiasi**, e da una **chiave** (di norma una stringa) che lo identifica univocamente rispetto agli altri elementi dello stesso dizionario.

In altre parole, un dizionario è un **insieme** di **coppie chiave–valore**. Tra di esse non è definito nessun ordinamento, e pertanto i loro valori sono accessibili solo attraverso la chiave corrispondente.

Le chiavi devono essere scelte dal programmatore, e possono essere valori numerici o Booleani, oppure stringhe. Per gli scopi di questo corso si considereranno solo **stringhe**. Come per le variabili, nella scelta delle chiavi è buona norma usare simboli mnemonici.

Il tipo di dato *dizionario*

Un dizionario si rappresenta nei programmi Python come una sequenza di coppie chiave–valore separate da virgole, racchiusa tra parentesi **graffe**; in ogni coppia, la chiave e il valore sono separati dal carattere `:`

Schematicamente: `{`chiave_1`:` valore_1`,` chiave_2`:` valore_2`,` `...` `}`

Il tipo di dato *dizionario*: esempi

- Un dizionario che contiene informazioni anagrafiche su una persona: nome e cognome, i cui valori sono stringhe, ed età, il cui valore è un numero intero; le chiavi sono le stringhe `"nome"`, `"cognome"`, `"età"`:
  `{"nome": "Maria", "cognome": "Bianchi", "età": 28}`
- Un dizionario che contiene le coordinate di un punto (due numeri frazionari) nel piano cartesiano, associate alle chiavi `"x"` e `"y"`:
  `{"x": -1.2, "y": 3.4}`
- Un dizionario che contiene una data (giorno, mese e anno, in formato numerico):
  `{"giorno": 1, "mese": 6, "anno": 2017}`
- un dizionario **vuoto**: `{}`

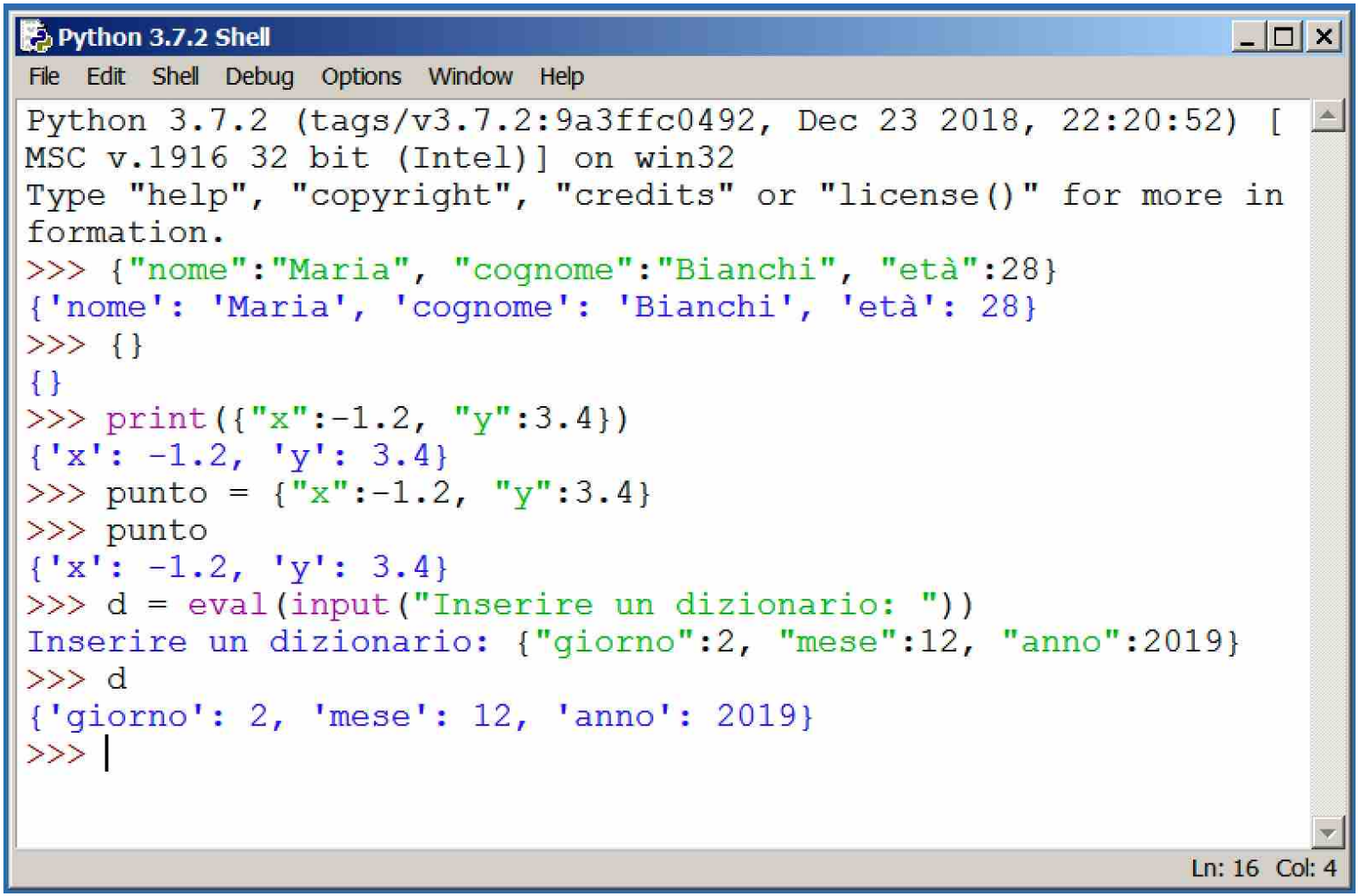
Il tipo di dato *dizionario*

Come per le liste e le stringhe, i valori del tipo *dizionario* sono **espressioni**. È quindi possibile

- scrivere un dizionario nella *shell*: dopo la pressione del tasto `INVIO`, il dizionario verrà mostrato nella stessa *shell*
- stampare un dizionario con l'istruzione `print`, per esempio:
  `print({"x": -1.2, "y": 3.4})`
- assegnare un dizionario a una variabile, per esempio:
  `punto = {"x": -1.2, "y": 3.4}`
- acquisire un dizionario attraverso la tastiera, per esempio:
  `d = eval(input("Inserire un dizionario: "))`

Nota: l'ordine con il quale vengono mostrate nella *shell* le coppie chiave–valore di un dizionario non può essere controllato dal programmatore.

Esempi


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [
MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more in
formation.
>>> {"nome":"Maria", "cognome":"Bianchi", "età":28}
{'nome': 'Maria', 'cognome': 'Bianchi', 'età': 28}
>>> {}
{}
>>> print({"x":-1.2, "y":3.4})
{'x': -1.2, 'y': 3.4}
>>> punto = {"x":-1.2, "y":3.4}
>>> punto
{'x': -1.2, 'y': 3.4}
>>> d = eval(input("Inserire un dizionario: "))
Inserire un dizionario: {"giorno":2, "mese":12, "anno":2019}
>>> d
{'giorno': 2, 'mese': 12, 'anno': 2019}
>>>
Ln: 16 Col: 4

Il tipo di dato *dizionario*

Così come per le liste, anche i valori degli elementi di un dizionario possono essere indicati attraverso espressioni.

Per esempio, dopo l'esecuzione delle seguenti istruzioni:

```
n = "Maria"
c1 = "Ros"
c2 = "si"
a = 5
persona = {"nome": n, "cognome": c1 + c2, "età": a**2}
```

la variabile `persona` sarà associata al seguente dizionario:

```
{"nome": "Maria", "cognome": "Rossi", "età": 25}
```

Il tipo di dato *dizionario*

Anche i valori degli elementi di un dizionario possono appartenere a un tipo qualsiasi, e quindi possono essere valori strutturati come stringhe, liste e dizionari (nidificati).

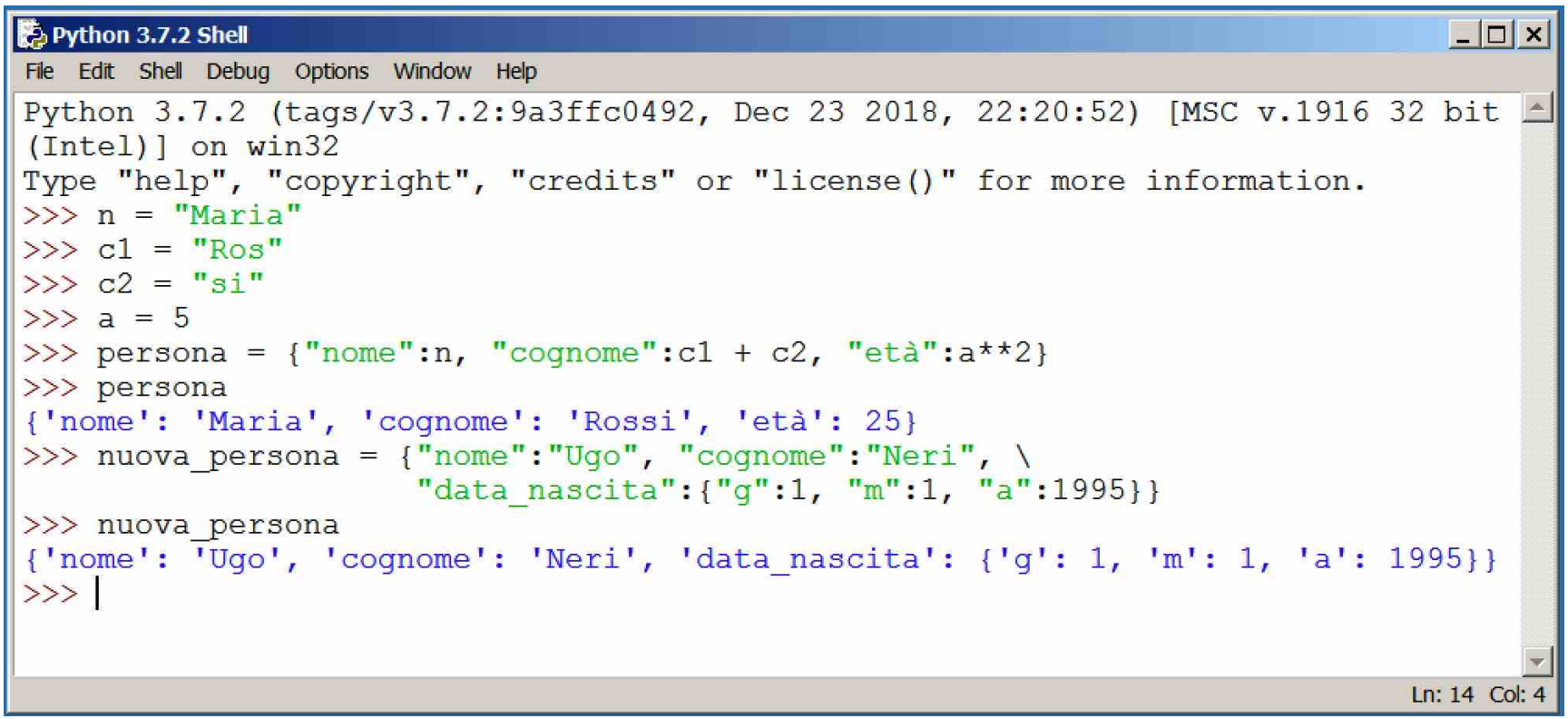
Per esempio, il dizionario seguente contiene il nome, il cognome e la data di nascita di una persona (giorno, mese e anno in formato numerico):

```
{"nome": "Ugo", "cognome": "Neri",\
 "g": 1, "m": 1, "a": 1995}
```

Le stesse informazioni possono essere memorizzate in una forma più strutturata mediante due dizionari nidificati:

```
{"nome": "Ugo", "cognome": "Neri", \
 "data_nascita": {"g": 1, "m": 1, "a": 1995}}
```

Esempi


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> n = "Maria"
>>> c1 = "Ros"
>>> c2 = "si"
>>> a = 5
>>> persona = {"nome":n, "cognome":c1 + c2, "età":a**2}
>>> persona
{'nome': 'Maria', 'cognome': 'Rossi', 'età': 25}
>>> nuova_persona = {"nome":"Ugo", "cognome":"Neri", \
"data_nascita":{"g":1, "m":1, "a":1995}}
>>> nuova_persona
{'nome': 'Ugo', 'cognome': 'Neri', 'data_nascita': {'g': 1, 'm': 1, 'a': 1995}}
>>>
Ln: 14 Col: 4

Il tipo di dato *dizionario*

Anche per i dizionari esistono diversi operatori e funzioni predefinite. In questo corso si vedranno solo gli operatori essenziali.

L'operatore principale è quello di **indicizzazione**: essp consente l'accesso ai **valori** dei singoli elementi.

Il nome e la sintassi sono identici a quelli dell'analogo operatore per le liste, con la differenza che gli elementi di una lista sono **ordinati** e quindi identificati univocamente dalla loro posizione (indice), mente gli elementi di un dizionario non hanno un ordine predefinito ma sono identificati dalla loro chiave.

Principali operatori sui dizionari

In sintesi, la sintassi degli operatori principali è la seguente

sintassi	**descrizione**
dizionario_1 == dizionario_2	confronto ("uguale a")
dizionario_1 != dizionario_2	confronto ("diverso da")
dizionario[chiave]	indicizzazione: accesso ai singoli elementi

Operatori di confronto

Gli operatori == e != consentono di scrivere espressioni **condizionali** (il cui valore sarà `True` o `False`) consistenti nel confronto tra due dizionari.

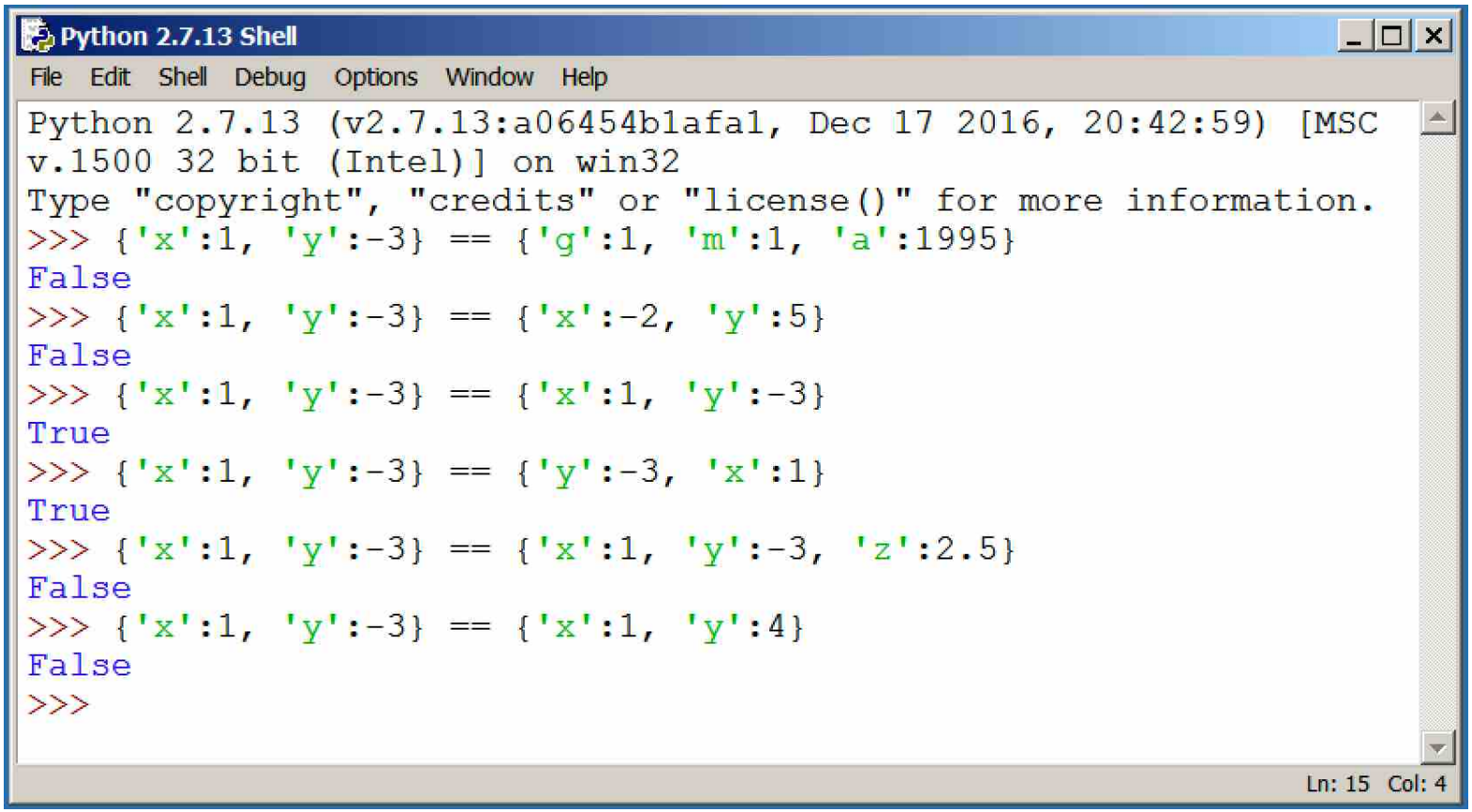
Sintassi

- dizionario_1 == dizionario_2
- dizionario_1 != dizionario_2

dove dizionario_1 e dizionario_2 indicano **espressioni** che abbiano come valore un dizionario.

Semantica: due dizionari sono considerati identici se contengono le stesse **coppie** chiave–valore, **indipendentemente** dal loro ordine.

Esempi


Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC
v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> {'x':1, 'y':-3} == {'g':1, 'm':1, 'a':1995}
False
>>> {'x':1, 'y':-3} == {'x':-2, 'y':5}
False
>>> {'x':1, 'y':-3} == {'x':1, 'y':-3}
True
>>> {'x':1, 'y':-3} == {'y':-3, 'x':1}
True
>>> {'x':1, 'y':-3} == {'x':1, 'y':-3, 'z':2.5}
False
>>> {'x':1, 'y':-3} == {'x':1, 'y':4}
False
>>>
Ln: 15 Col: 4

L'operatore di indicizzazione

L'operatore di **indicizzazione** consente di accedere al **valore** di un elemento di un dizionario, per mezzo della **chiave** corrispondente.

Sintassi: dizionario[chiave]

- dizionario deve essere un'espressione che produca un dizionario (di norma, il nome di una variabile)
- chiave deve essere un'espressione il cui valore corrisponda a una delle chiavi del dizionario

Semantica: il risultato è il valore dell'elemento di dizionario associato a chiave. Se il valore di chiave non corrisponde a una delle chiavi del dizionario si otterrà un messaggio d'errore.

L'operatore di indicizzazione

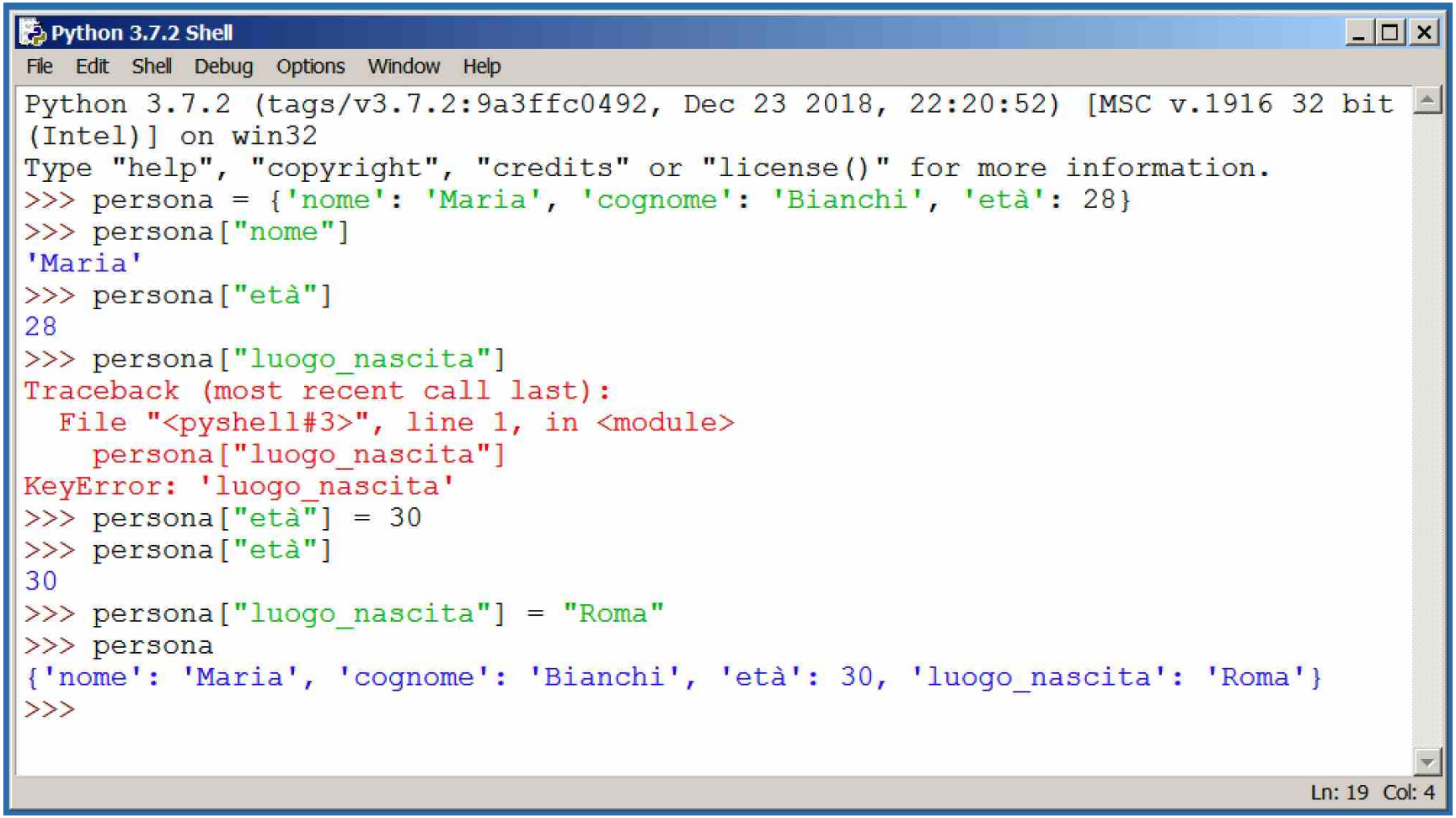
L'operatore di indicizzazione consente anche di

- **modificare** il valore associato a una chiave **esistente**
- aggiungere una **nuova** coppia chiave–valore, la cui chiave **non** sia presente nel dizionario

Sintassi: dizionario[chiave] = valore

Semantica: se chiave fa parte di dizionario, il valore corrispondente viene **sostituito** da valore; altrimenti viene **aggiunto** al dizionario l'elemento chiave:valore

Esempi


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> persona = {'nome': 'Maria', 'cognome': 'Bianchi', 'età': 28}
>>> persona["nome"]
'Maria'
>>> persona["età"]
28
>>> persona["luogo_nascita"]
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
persona["luogo_nascita"]
KeyError: 'luogo_nascita'
>>> persona["età"] = 30
>>> persona["età"]
30
>>> persona["luogo_nascita"] = "Roma"
>>> persona
{'nome': 'Maria', 'cognome': 'Bianchi', 'età': 30, 'luogo_nascita': 'Roma'}
>>>
Ln: 19 Col: 4

Esempio: costruzione di un dizionario

In molti programmi è necessario costruire un dizionario avente un insieme di chiavi **predefinito** (ovvero noto al programmatore), alle quali dovranno però essere associati valori acquisiti o calcolati durante l'**esecuzione** del programma.

Questo si può ottenere attraverso l'operatore di indicizzazione, a partire da un dizionario vuoto.

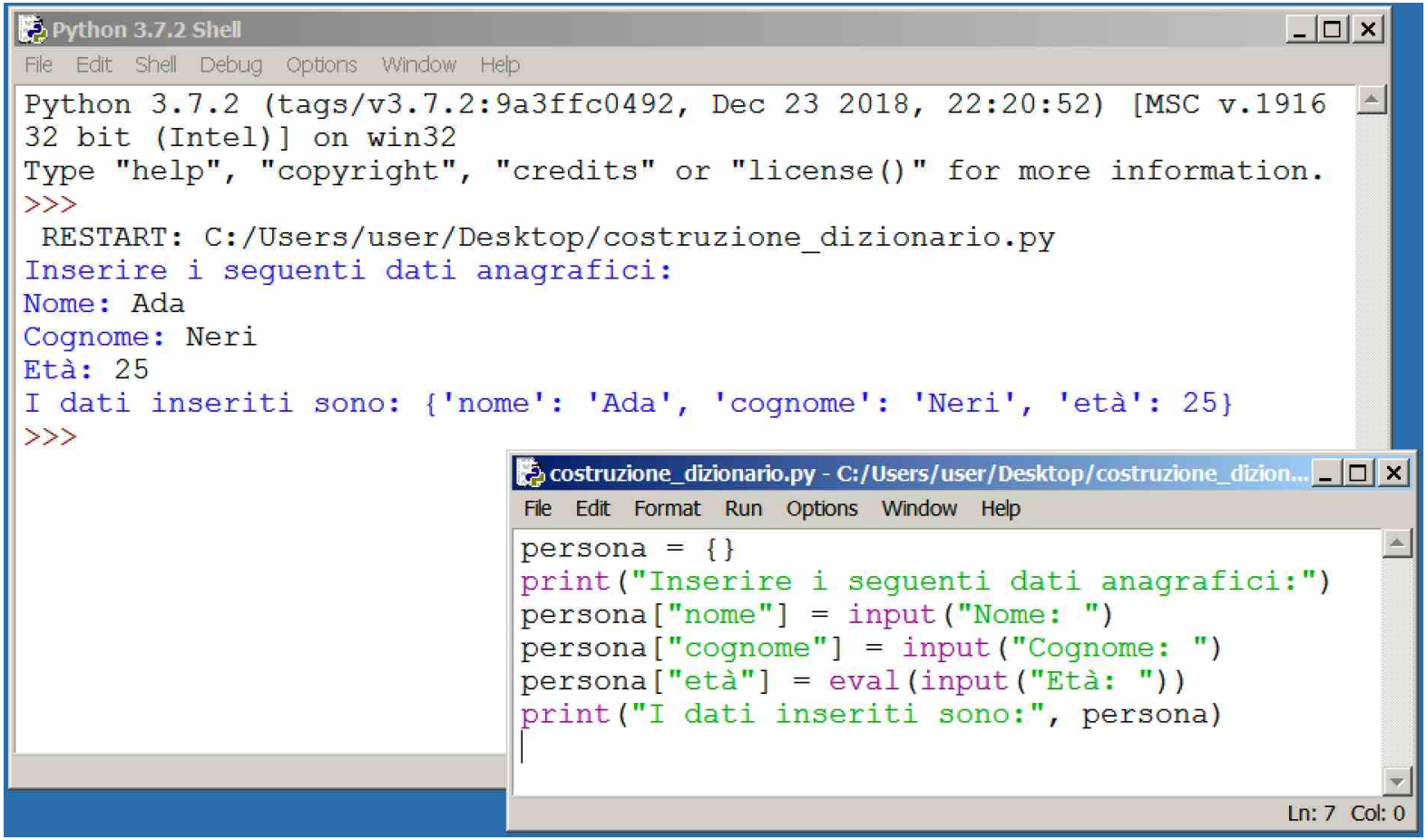
Esempio: costruzione di un dizionario

Per esempio, si assuma di voler memorizzare in un dizionario il nome, il cognome e l'età (un numero intero) di una persona, che dovranno essere acquisiti dal programma attraverso la tastiera.

A questo scopo si potrà usare un dizionario con tre chiavi: `"nome"`, `"cognome"` ed `"età"`. Il programma, disponibile nel *file* `58_costruzione_dizionario.py`, è il seguente:

```
persona = {}
print("Inserire i seguenti dati anagrafici:")
persona["nome"] = input("Nome: ")
persona["cognome"] = input("Cognome: ")
persona["età"] = eval(input("Età:  "))
print("I dati inseriti sono:", persona)
```

Esempio: costruzione di un dizionario

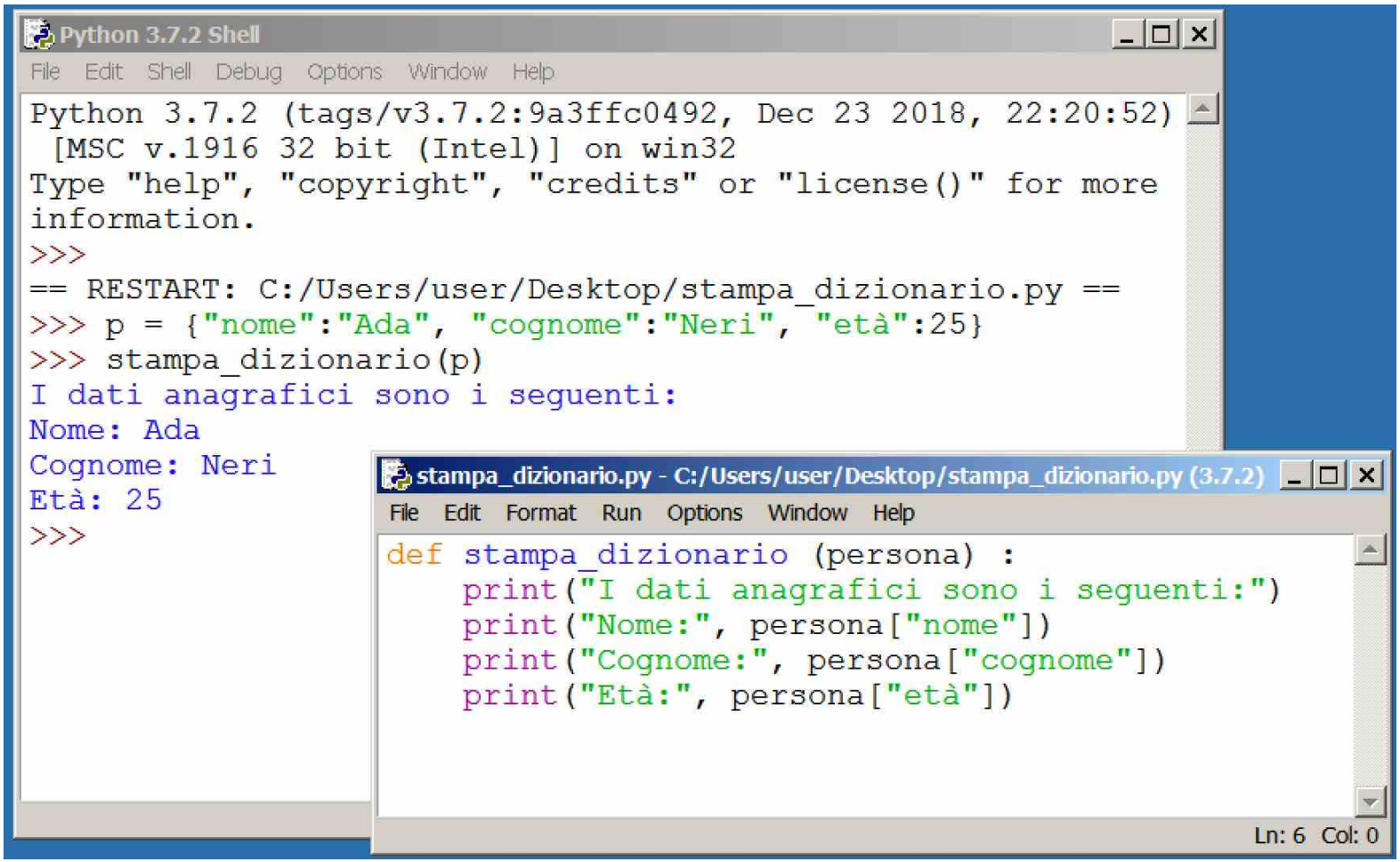


Esempio: accesso agli elementi di un dizionario

Una funzione che riceve come argomento un dizionario avente la stessa struttura considerata nell'esercizio precedente, e contenente informazioni anagrafiche su una persona; la funzione stampa tali informazioni nella *shell*, precedute da opportuni messaggi (*file*: `59_stampa_dizionario.py`)

```
def stampa_dizionario(persona):
    print("I dati anagrafici sono i seguenti:")
    print("Nome:", persona["nome"])
    print("Cognome:", persona["cognome"])
    print("Età:", persona["età"])
```

Esempio: accesso agli elementi di un dizionario

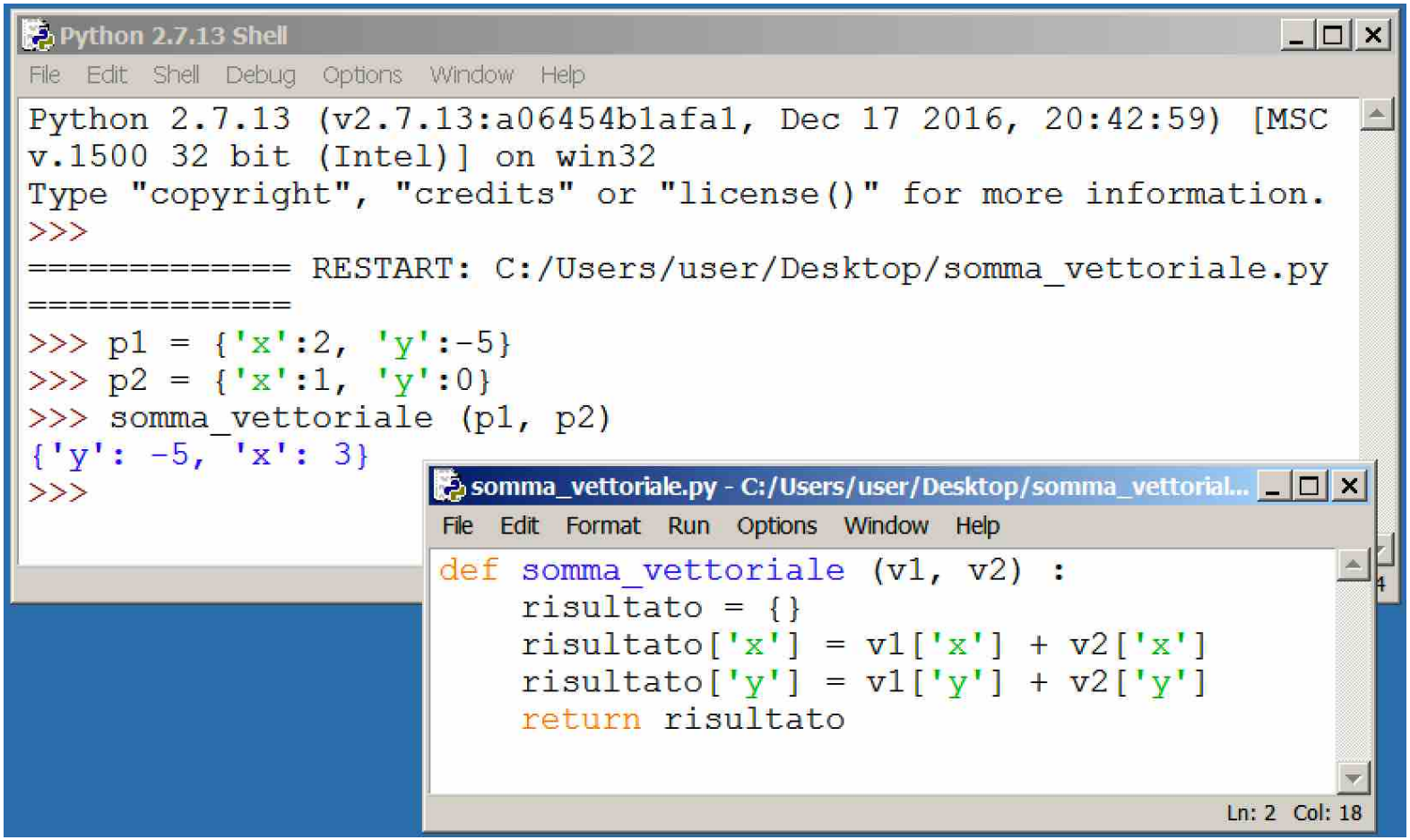


Esempio: somma vettoriale

In un esempio precedente (*file*: `49_somma_vettoriale.py`) si è definita una funzione per calcolare la somma di due vettori le cui componenti siano memorizzate in due liste.
Nella funzione che segue (*file*: `60_somma_vettoriale.py`) si realizza la stessa operazione su vettori a due dimensioni le cui componenti siano memorizzate in due dizionari aventi per chiavi le stringhe `"x"` e `"y"`.

```
def somma_vettoriale(v1, v2):
    risultato = {}
    risultato["x"] = v1["x"] + v2["x"]
    risultato["y"] = v1["y"] + v2["y"]
    return risultato
```

Esempio: somma vettoriale



Accesso agli elementi di strutture nidificate in un dizionario

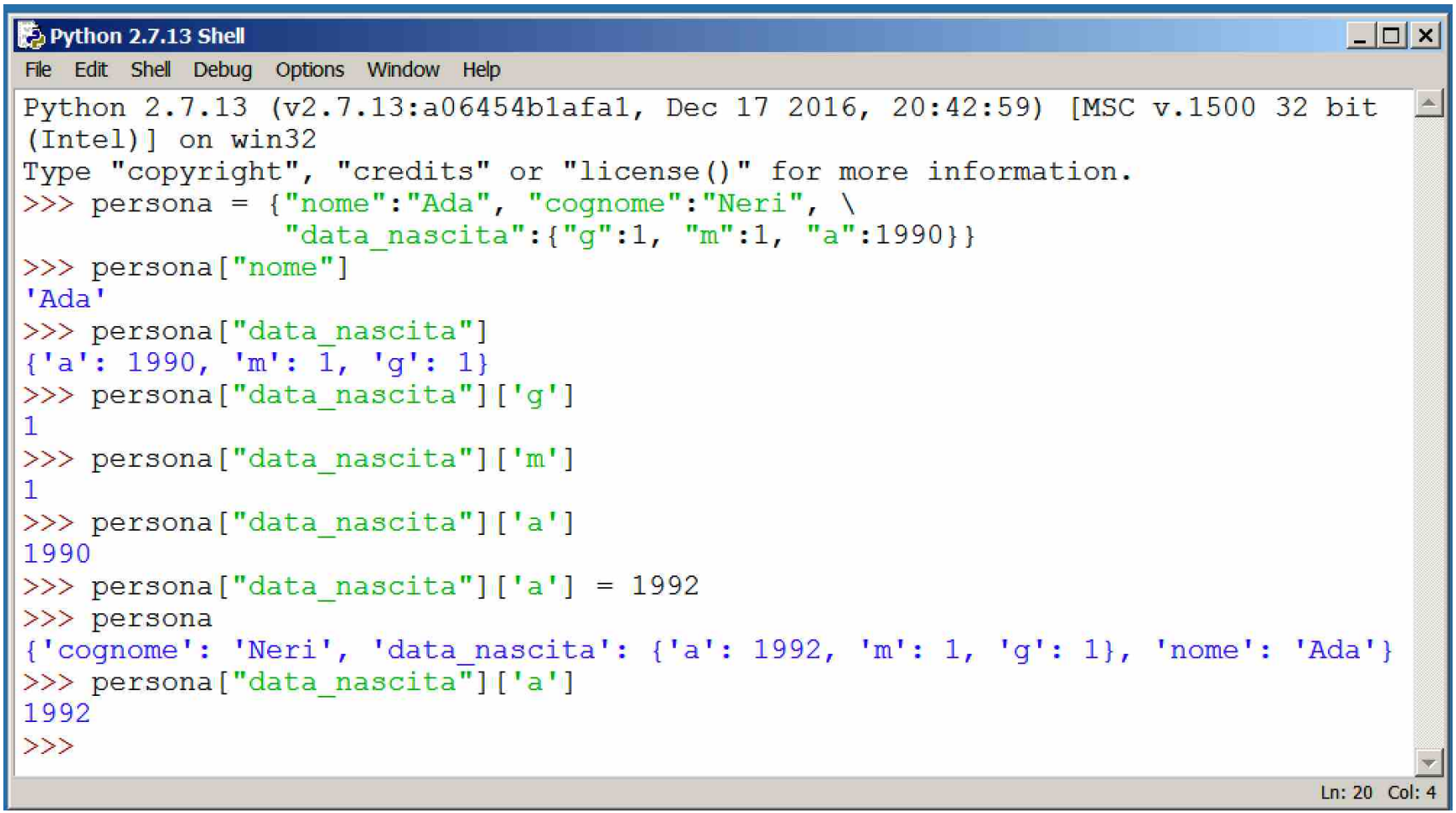
Si è già detto che anche i valori associati alle chiavi di un dizionario possono essere di tipo qualsiasi, quindi possono essere a loro volta dati strutturati, cioè stringhe, liste e dizionari.

I valori strutturati contenuti in un dizionario sono accessibili attraverso l'operatore di indicizzazione, con la stessa sintassi già descritta per il caso di strutture nidificate nelle liste.

In particolare, se d è una variabile a cui è stato associato un dizionario, e l'elemento d[chiave] è un valore strutturato (una sequenza o un altro dizionario), è possibile accedere all'elemento avente indice (nel caso di una sequenza) o chiave (se si tratta di un dizionario) k, con la seguente **sintassi**:

```
d[chiave][k]
```

Esempi


Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> persona = {"nome":"Ada", "cognome":"Neri", \
"data_nascita":{"g":1, "m":1, "a":1990}}
>>> persona["nome"]
'Ada'
>>> persona["data_nascita"]
{'a': 1990, 'm': 1, 'g': 1}
>>> persona["data_nascita"]['g']
1
>>> persona["data_nascita"]['m']
1
>>> persona["data_nascita"]['a']
1990
>>> persona["data_nascita"]['a'] = 1992
>>> persona
{'cognome': 'Neri', 'data_nascita': {'a': 1992, 'm': 1, 'g': 1}, 'nome': 'Ada'}
>>> persona["data_nascita"]['a']
1992
>>>
Ln: 20 Col: 4

Esempi

Si vuole scrivere un programma che acquisisca il nome, il cognome, la data e il luogo di nascita di una persona, e li memorizzi in un dizionario avente chiavi `"nome"`, `"cognome"`, `"luogo_nascita"` e `"data_nascita"`. La data di nascita deve essere a sua volta memorizzata (in formato numerico) in un dizionario avente chiavi `"giorno"`, `"mese"` e `"anno"`.

Il programma mostrato di seguito è disponibile nel *file* `61_dati_anagrafici.py`

Esempi

```
persona = {}
print("Inserire i seguenti dati anagrafici:")
persona["nome"] = input("Nome: ")
persona["cognome"] = input("Cognome: ")
persona["luogo_nascita"] = input("Luogo di nascita: ")
persona["data_nascita"] = {}
print("Data di nascita:")
persona["data_nascita"]["giorno"] = eval(input("giorno: "))
persona["data_nascita"]["mese"] = eval(input("mese: "))
persona["data_nascita"]["anno"] = eval(input("anno: "))
print("I dati immessi sono i seguenti:")
print(persona)
```

Esempi

Si vuole ora modificare il programma precedente in modo che sia in grado di acquisire le stesse informazioni su un **insieme** di persone (il cui numero dovrà prima essere chiesto all'utente), costruendo un dizionario distinto per ogni persona.

In questo caso è conveniente memorizzare l'insieme dei dizionari all'interno di una lista, che sarà costruita iterativamente per concatenazione a partire da una lista vuota.

Il programma mostrato di seguito è disponibile nel *file* `62_dati_anagrafici.py`

Esempi

```
persone = []
n_persone = eval(input("Quante persone? "))
n = 1
while n <= n_persone:
   persona = {}
   print("Dati anagrafici della persona n. " + str(n) + ":")
   persona["nome"] = input(" nome: ")
   persona["cognome"] = input(" cognome: ")
   persona["luogo_nascita"] = input(" luogo di nascita: ")
   persona["data_nascita"] = {}
   print(" data di nascita:")
   persona["data_nascita"]["giorno"] = eval(input(" giorno: "))
   persona["data_nascita"]["mese"] = eval(input(" mese: "))
   persona["data_nascita"]["anno"] = eval(input(" anno: "))
   persone = persone + [persona]
   n = n + 1
print("I dati immessi sono i seguenti:")
print(persone)
```

Esempi

Un esempio analogo al precedente: si vuole scrivere un programma che acquisisca le coordinate di un insieme di punti del piano cartesiano. Le coordinate di ciascun punto saranno memorizzate in un dizionario avente chiavi `"x"` e `"y"`, e i dizionari saranno a loro volta memorizzati all'interno di una lista.

Il programma mostrato di seguito è disponibile nel *file* `63_punti.py`

Esempi

```
n_punti = eval(input("Numero di punti: "))
punti = []
n = 1
while n <= n_punti:
    punto = {}
    print("Coordinate del punto n. " + str(n) + ":")
    punto["x"] = eval(input("x: "))
    punto["y"] = eval(input("y: "))
    punti = punti + [punto]
    n = n + 1
print("Sono stati inseriti i seguenti punti:")
print(punti)
```

Esempi

In questo esempio si vuole memorizzare in un dizionario l'esito di una gara di salto in alto di un singolo atleta: nome, cognome e numero di gara dell'atleta, e misura (espressa in cm) di ogni salto valido; gli eventuali salti non validi dovranno essere codificati con il valore 0.

Poiché il numero di prove in una gara di salto in alto non è definito a priori ma dipende dall'esito di ciascuna prova (e può variare da atleta ad atleta), è conveniente memorizzare in una lista la sequenza delle misure. Il numero delle prove dovrà prima essere chiesto all'utente.

La lista con le misure di ciascuna prova può a sua volta essere memorizzata all'interno di un dizionario che conterrà altri tre valori, corrispondenti al nome, al cognome e al numero di gara dell'atleta.

Per esempio, nel caso di un atleta di nome Marco Bianchi, numero di gara 24, che abbia eseguito sei prove delle quali tre valide (con misure 180, 185 e 188 cm) e tre nulle (la seconda, la terza e la sesta), il dizionario da costruire sarà:

```
{"nome": "Marco", "cognome": "Bianchi", "numero": 24,
 "prove": [180, 0, 0, 185, 188, 0]}
```

Esempi

Il programma seguente è disponibile nel *file* `64_salto_in_alto.py`

```
atleta = {}
atleta["nome"] = input("Nome dell'atleta: ")
atleta["cognome"] = input("Cognome: ")
atleta["numero"] = eval(input("Numero di gara: "))
atleta["prove"] = []
n_prove = eval(input("Numero di prove: "))
print("Inserire la misura di ciascuna prova")
print("   (0 per le prove nulle):")
n = 1
while n <= n_prove:
    misura = eval(input(" prova n. " + str(n) + ": "))
    atleta["prove"] = atleta["prove"] + [misura]
    n = n + 1
print("I dati immessi sono i seguenti:")
print(atleta)
```

Esempi

Se si volessero memorizzare i dati su tutti gli atleti che hanno partecipato a una gara di salto in alto, i dizionari corrispondenti a ciascun atleta potrebbero essere memorizzati all'interno di una lista (in modo analogo a quanto già visto in un esempio precedente).

Il programma corrispondente è disponibile nel *file* `65_gara_salto_in_alto.py`

Esempi

In quest'ultimo esempio si vogliono memorizzare gli esiti di un esame sostenuto da un insieme di studenti. Per ogni studente si dovrà memorizza il nome, il cognome, il numero di matricola (sotto forma di una stringa) e il voto. Il voto dovrà essere rappresentato come numero intero: 0 nel caso di esito insufficiente, oppure un valore tra 18 e 31, dove 31 corrisponde a 30 con lode.

Anche in questo caso si potrà usare per ogni singolo studente un dizionario con quattro chiavi (corrispondenti a nome, cognome, matricola e voto). Un esempio:

```
{"nome": "Marco", "cognome": "Bianchi",
 "matricola": "12345", "voto": 28}
```

L'insieme dei dizionari verrà memorizzato all'interno di una lista.

Esempi

Il programma seguente è disponibile nel *file* `66_esame.py`

```
studenti = []
n_studenti = eval(input("Quanti studenti? "))
n = 1
while n <= n_studenti:
    studente = {}
    print("Dati dello studente n. " + str(n) + ":"))
    studente["nome"] = input(" nome: ")
    studente["cognome"] = input(" cognome: ")
    studente["matricola"] = input(" matricola: ")
    studente["voto"] = eval(input(" voto: "))
    studenti = studenti + [studente]
    n = n + 1
print("I dati inseriti sono i seguenti:")
print(studenti)
```

Definizione di strutture dati

La scrittura di un programma ha come presupposto due fasi fondamentali

- la formulazione di un algoritmo
- la scelta delle **strutture dati** per rappresentare i dati da elaborare

Nella seconda fase le **strutture dati** possono essere descritte in astratto senza fare riferimento ai tipi di dato di uno specifico linguaggio, per esempio in termini di sequenze ordinate di numeri o caratteri, di collezioni di valori associati a "etichette" simboliche, oppure di strutture più complesse come matrici e grafi.

Definizione di strutture dati

Quando si passa alla codifica di un algoritmo in un dato linguaggio di programmazione, le strutture dati devono essere "tradotte" nei tipi di dato (semplici e strutturati) disponibili in tale linguaggio.

Per esempio, si è già visto che in linguaggio Python le sequenze ordinate di valori possono essere rappresentate attraverso liste (o attraverso stringhe nel caso particolare di sequenze di caratteri), mentre le collezioni di valori associabili a "etichette" simboliche possono essere rappresentate mediante dizionari.

Definizione di strutture dati

La scelta delle strutture dati più opportune è un passo fondamentale, è può influenzare la complessità di un programma. Una scelta errata può persino pregiudicare la possibilità di codificare un dato algoritmo in un programma eseguibile dal calcolatore.

Di seguito si mostrano alcuni esempi concreti della scelta delle strutture dati, facendo riferimento agli esempi di programmi mostrati in precedenza.

Definizione di strutture dati: esempi

Se in un programma si volessero memorizzare ed elaborare le coordinate di punti in uno spazio di una data dimensione, la soluzione più immediata sarebbe usare una variabile semplice (di tipo `float`) per ogni coordinata di ogni singolo punto.

Per esempio, le coordinate di un singolo punto nel piano cartesiano potrebbero essere memorizzate in due variabili di nome `x` e `y`.

L'inconveniente di questa scelta consiste nel fatto che le due variabili, pur essendo associate a valori relativi a una **singola** entità logica (un punto), non hanno tra loro nessun "legame" all'interno di un programma. In altre parole, il programmatore (o chi leggesse il programma) dovrebbe ricordare che le due variabili `x` e `y` si riferiscono a una stessa entità logica.

Definizione di strutture dati: esempi

Se si volessero memorizzare le coordinate di $n > 1$ punti in uno spazio di dimensione d, e i valori di n e d fossero noti al programmatore **nel momento in cui scrive il programma**, la scelta precedente richiederebbe l'uso di un numero di variabili pari a $n \times d$.

Per esempio, nel caso $n = 3$ e $d = 2$ una scelta ragionevole per i nomi delle variabili è la seguente: `x1` e `y1` (primo punto), `x2` e `y2` (secondo punto), `x3` e `y3` (terzo punto).

Definizione di strutture dati: esempi

Se la dimensione dello spazio o il numero di punti fosse molto grande, è facile rendersi conto che l'uso di variabili distinte per ogni coordinata di ogni punto porterebbe a programmi di lunghezza e complessità eccessiva.

In particolare, non sarebbe possibile usare istruzioni iterative per acquisire, elaborare o stampare i valori delle coordinate di un punto, o dei vari punti.

Se invece la dimensione dello spazio o il numero di punti non fossero noti **nel momento in cui si scrive il programma**, l'uso di variabili distinte **non sarebbe possibile**, poiché non sarebbe noto il numero di variabili da usare.

Definizione di strutture dati: esempi

Per rappresentare ciascun punto è più opportuno usare un tipo strutturato: in questo modo i valori delle coordinate possono essere memorizzati in una **singola** variabile.

Uno dei vantaggi di questa scelta consiste nel rendere evidente al programmatore, e a chiunque leggesse il programma, il fatto che gli elementi di una tale variabile sono componenti di un'unica entità logica.

Definizione di strutture dati: esempi

A questo scopo si è già visto che in linguaggio Python è possibile usare sia liste che dizionari.

Per esempio, le coordinate del punto $(3, -1)$ potrebbero essere memorizzate in una **singola** variabile di nome `punto`, il cui valore potrà essere

- la lista `[3, -1]`: in questo modo le coordinate sarebbero accessibili con la sintassi `punto[0]` e `punto[1]`
- il dizionario `{"x": 3, "y": -1}`: in questo modo le coordinate sarebbero accessibili con la sintassi `punto["x"]` e `punto["y"]`

Definizione di strutture dati: esempi

Dal punto di vista della leggibilità e della facilità di comprensione di un programma, nel caso delle coordinate di un punto l'uso di un dizionario può essere preferibile a una lista, poiché un dizionario consente l'uso di simboli convenzionali come `x` e `y` per le chiavi.

Tuttavia, nel caso di spazi a più di tre dimensioni l'uso dei dizionari non sarebbe conveniente, poiché in questo caso la notazione convenzionale usata in matematica non prevede l'uso di simboli per indicare le componenti di un vettore. La scelta delle chiavi sarebbe quindi arbitraria; per es., nel caso di uno spazio a cinque dimensioni ai simboli `x`, `y` e `z` si potrebbero aggiungere `p` e `q`, ma questo non faciliterebbe la comprensione di un'espressione come `punto["q"]` (si tratta della quarta o della quinta componente?).

Definizione di strutture dati: esempi

Nel caso di spazi a più di tre dimensioni, le liste sono una scelta più opportuna.

La notazione matematica convenzionale prevede infatti l'uso di pedici (o apici) numerici per indicare le componenti di un vettore, per esempio p_5 indica la quinta componente.

Analogamente, una lista consentirebbe di riferirsi al quinto elemento con la sintassi `punto[4]`.

Definizione di strutture dati: esempi

Per memorizzare una collezione **predefinita** di valori relativi a una stessa entità, ciascuno dei quali abbia un significato descrivibile con una "etichetta" simbolica, è preferibile usare un dizionario.

Alcuni esempi, già visti in precedenza, sono le informazioni anagrafiche su una persona (per es., nome, cognome, data e luogo di nascita e codice fiscale), oppure i dati su uno studente che ha sostenuto un certo esame (per es., nome, cognome, matricola, data dell'esame e voto conseguito).

Definizione di strutture dati: esempi

L'uso di un dizionario consente infatti **al programmatore** di ricordare facilmente il significato di ciascun elemento attraverso la chiave corrispondente, purché si usino chiavi mnemoniche, come `"nome"`, `"voto"`, ecc.

L'uso di una lista richiede invece al programmatore lo sforzo aggiuntivo di ricordare a quale componente corrisponde ogni posizione della lista.

Definizione di strutture dati: esempi

Per esempio, si considerino le due alternative seguenti:

```
persona_a = {"nome":"Ada", "cognome":"Neri", "età":25}
persona_b = ["Ada", "Neri", 25]
```

È evidente che per accedere al cognome di una persona l'espressione:

```
persona_a["cognome"]
```

è più comprensibile rispetto a:

```
persona_b[1]
```

Definizione di strutture dati: esempi

Si consideri ora il caso in cui si debba memorizzare una sequenza o una collezione (non ordinata) di valori, la cui dimensione **non** sia nota al programmatore **nel momento in cui scrive il programma**.

Per esempio, questo può accadere in un programma che debba elaborare

- vettori (punti) in uno spazio di dimensione **qualsiasi**, come si è già visto in un esempio precedente
- i dati su un insieme di studenti che abbiano sostenuto un certo esame
- i dati su un insieme di atleti che abbiano partecipato a una certa gara

Definizione di strutture dati: esempi

In questo caso è preferibile usare una lista, anche se tra i valori che si devono elaborare **non** esiste nessun ordinamento predefinito (come nel caso di un insieme di studenti o di atleti).

L'uso di un dizionario richiederebbe infatti al programmatore la scelta di una chiave distinta per ciascun valore da memorizzare al suo interno, ma in casi come quelli degli esempi precedenti non esistono scelte delle chiavi che rendano più semplice la comprensione del significato o del ruolo degli elementi di un dizionario rispetto agli indici di una lista.

Si ripensi per esempio al caso delle coordinate di un punto in uno spazio a più di tre dimensioni, discusso in precedenza.

Definizione di strutture dati: esempi

Per capire meglio questo punto, si consideri ancora il caso dei dati su un esame sostenuto da un insieme di studenti.

Se si volessero memorizzare i dati di ogni studente come elementi di un dizionario (a loro volta contenuti in un dizionario con chiavi come `"nome"`, `"matricola"`, ecc.), si dovrebbero usare chiavi come `"studente1"`, `"studente2"`, ecc. Per esempio:

```
{"studente1": {"matricola":"12345", "nome": "Luca", ... },
 "studente2": {"matricola":"54321", "nome": "Ugo", ...  } }
```

Se l'intero dizionario fosse memorizzato in una variabile di nome `studenti`, i dati di ogni studente sarebbero accessibili mediante espressioni come `studenti["studente2"]`.

Definizione di strutture dati: esempi

Questa soluzione non è però più mnemonica di quella richiesta per l'accesso agli elementi di una lista, per esempio `studenti[1]`.

Inoltre l'uso di una lista consente l'accesso in sequenza a tutti i suoi elementi con un'istruzione iterativa, per esempio:

```
for studente in studenti: ...
```

oppure:

```
k = 0
while k < len(studenti): ...
```

Ciò è possibile anche per un dizionario usando gli elementi del linguaggio Python visti in questo corso, anche se al costo di una maggiore complessità. In ogni caso, questa possibilità non offre vantaggi rispetto all'uso delle liste dal punto di vista della facilità di comprensione di un programma.

Dizionari come valori *mutabili*

Analogamente alle liste, anche i dizionari sono valori **mutabili**: come si è visto, è infatti possibile modificare i loro elementi con l'indicizzazione e l'assegnamento.

Per questo motivo

- se alla variabile `x` è stato associato un dizionario, l'assegnamento `y = x` fa sì che `y` sia associata allo **stesso** dizionario; come conseguenza, ogni modifica agli elementi del dizionario eseguita attraverso `x` o `y` si rifletterà anche sull'altra variabile
- se si passa un dizionario a una funzione come argomento, ogni modifica eseguita sul dizionario associato al parametro della funzione si riflette anche sul dizionario del programma chiamante

Esempi

- Dopo l'esecuzione delle istruzioni

```
p = {"x": 1, "y"': -2.5}
q = p
p["x"] = 0
```

  il valore associato a `p` e `q` sarà `{"x": 0, "y": -2.5}`

- Si consideri la seguente funzione:

```
def azzera_punto(p):
    p["x"] = 0.0
    p["y"] = 0.0
```

  Se, dopo la definizione della funzione `azzera_punto`, si eseguono le istruzioni

```
punto = {"x": 1, "y": -2.5}
azzera_punto(punto)
```

  il valore associato a `punto` sarà `{"x": 0.0, "y": 0.0}`

Copia di un dizionario

Si è visto che per eseguire una **copia** di una lista è possibile usare l'operatore di *slicing*.

Per eseguire una copia di un dizionario è invece possibile usare la funzione predefinita `copy`, che prevede la seguente sintassi:

dizionario`.copy()`

Esempi

- Dopo l'esecuzione delle istruzioni

```
p = {"x": 1, "y": -2.5}
q = p.copy()
p["x"] = 0
```

  il valore associato a `p` sarà `{"x": 0, "y": -2.5}`, mentre quello associato a `q` sarà ancora `{"x": 1, "y": -2.5}`

- Si consideri ancora la funzione `azzera_punto` dell'esempio precedente. Dopo l'esecuzione delle istruzioni

```
punto = {"x": 1, "y": -2.5}
azzera_punto(punto.copy())
```

  il valore associato alla variabile `punto` sarà ancora

```
{"x": 1, "y": -2.5}
```

Algoritmi di ricerca e ordinamento

Problemi di ricerca e di ordinamento

Due problemi che ricorrono in molte applicazioni pratiche consistono nella **ricerca** di un certo valore in una sequenza, e nell'**ordinamento** di un insieme di valori secondo un certo criterio.

Di seguito si descrivono alcuni semplici algoritmi di ricerca e ordinamento, e si mostra una loro codifica in linguaggio Python.

Per semplicità si considereranno problemi che coinvolgono **liste di numeri**; gli algoritmi presentati sono però applicabili a dati di qualsiasi tipo.

Algoritmo di ricerca sequenziale

Si consideri la formulazione di un algoritmo per determinare se un certo numero x sia presente in una data sequenza di numeri S di lunghezza qualsiasi.

Questo problema richiede una risposta binaria: "vero" (se x è presente in S) oppure "falso".

Un semplice algoritmo consiste nell'analizzare gli elementi di S **dal primo all'ultimo**, confrontando ciascuno di essi con x. Se l'elemento in esame coincide con x l'algoritmo termina con la risposta "vero"; se **tutti** gli elementi di S sono già stati analizzati (e quindi nessuno di essi coincide con x) l'algoritmo termina con la risposta "falso".

Tale algoritmo è detto, per ragioni facilmente intuibili, **ricerca sequenziale**.

Algoritmo di ricerca sequenziale

Il numero di confronti eseguiti dall'algoritmo di ricerca sequenziale dipende sia dalla specifica sequenza che dal valore da cercare all'interno di essa.

Nel caso "migliore" x corrisponde al **primo** elemento di S: in questo caso viene eseguito **un solo** confronto.

Nel caso "peggiore" x si trova nell'**ultima** posizione di S, oppure non è presente in essa: in entrambi i casi il numero di confronti è pari al numero di elementi di S.

Algoritmo di ricerca sequenziale

L'algoritmo di ricerca sequenziale può essere codificato in Python mediante una funzione che riceve due argomenti: una lista contenente una sequenza di numeri e il numero da cercare. La funzione restituirà il valore `True` oppure `False`.

A questo scopo si esegue un'iterazione sugli elementi della lista, dal primo all'ultimo, per confrontare ciascuno di essi con il valore cercato. Se i due valori confrontati sono identici, la funzione può terminare immediatamente restituendo il valore `True`. In questo modo, se il valore cercato non è presente nella lista l'iterazione verrà portata a termine; ne consegue che in tal caso la funzione dovrà terminare restituendo il valore `False`.

Algoritmo di ricerca sequenziale

Due versioni della stessa funzione che fanno uso delle istruzioni `while` e `for` (si veda il *file* `67_ricerca_sequenziale.py`)

```
def ricerca_sequenziale_1(sequenza, valore):
    k = 0
    while k < len(sequenza):
        if sequenza[k] == valore:
            return True
        k = k + 1
    return False

def ricerca_sequenziale_2(sequenza, valore):
    for elemento in sequenza:
        if elemento == valore:
            return True
    return False
```

Algoritmo di ricerca binaria

Se gli elementi della sequenza sono **ordinati** in senso crescente o decrescente si può formulare un algoritmo più efficiente.

Tale algoritmo si basa su un procedimento analogo a quello che si seguirebbe nella ricerca della pagina contenente un dato nome in un elenco telefonico. Informalmente: si apre l'elenco in corrispondenza delle due pagine centrali; se il nome cercato si trova in tali pagine la ricerca (della pagina) termina; altrimenti:

- se il nome cercato **precede** (in ordine alfabetico) quelli presenti nelle pagine in esame, la ricerca prosegue **in modo analogo** nelle pagine **precedenti** (aprendo cioè l'elenco a metà di tali pagine, ecc.)
- se il nome cercato **segue** quelli nelle pagine considerate, la ricerca procede in modo analogo nelle pagine **successive**

Da qui il nome di algoritmo di **ricerca binaria**.

Algoritmo di ricerca binaria

Per una descrizione più rigorosa si consideri una sequenza S di lunghezza qualsiasi, composta da numeri disposti in ordine **crescente**, e un valore x da cercare al suo interno.

Si inizia confrontando x con l'elemento in posizione **centrale** di S (si indichi con c tale elemento); se $x = c$ la ricerca termina e la risposta è "vero"; in caso contrario:

- se $x < c$, la ricerca procede **in modo analogo** nella sottosequenza di S che **precede** c
- se $x > c$, la ricerca procede **in modo analogo** nella sottosequenza di S che **segue** c

È facile rendersi conto che se x non fa parte di S, dopo un numero **finito** di confronti la sottosequenza da analizzare sarà **vuota**: in questo caso l'algoritmo terminerebbe producendo il risultato "falso".

Nota: detto N il numero di elementi della sequenza da analizzare in un passo qualsiasi dell'algoritmo, se N è **pari** si può considerare come elemento centrale quello in posizione $\frac{N}{2}$ oppure $\frac{N}{2} + 1$.

Algoritmo di ricerca binaria: esempio

Determinare se il numero 31 sia presente nella sequenza

$$1,\ 6,\ 8,\ 10,\ 18,\ 25,\ 32,\ 34,\ 37,\ 52,\ 60$$

Di seguito si mostra la sottosequenza considerata in ogni passo dell'algoritmo e, in rosso, l'elemento centrale

- primo passo: $1, 6, 8, 10, 18, \color{red}{25}, 32, 34, 37, 52, 60$
  $\color{red}{25} < 31$: si prosegue con la sottosequenza a destra
- secondo passo: $32, 34, \color{red}{37}, 52, 60$
  $\color{red}{37} > 31$: si prosegue con la sottosequenza a sinistra
- terzo passo: $\color{red}{32}, 34$
  $\color{red}{32} > 31$: si prosegue con la sottosequenza a sinistra
- quarto passo: la sottosequenza è vuota

Risultato: l'elemento cercato non è presente.

Algoritmo di ricerca binaria: esempio

Determinare se il numero 8 sia presente nella sequenza

$$1,\ 6,\ 8,\ 10,\ 18,\ 25,\ 32,\ 34,\ 37,\ 52,\ 60$$

- primo passo: $1, 6, 8, 10, 18, \mathbf{25}, 32, 34, 37, 52, 60$
  $25 > 8$: si prosegue con la sottosequenza a sinistra
- secondo passo: $1, 6, \mathbf{8}, 10, 18$
  $8 = 8$: l'algoritmo termina

Risultato: l'elemento cercato è presente.

Algoritmo di ricerca binaria: esempio

Determinare se il numero 6 sia presente nella sequenza

$$1,\ 6,\ 8,\ 10,\ 18,\ 25,\ 32,\ 34,\ 37,\ 52,\ 60$$

- primo passo: $1, 6, 8, 10, 18, 25, 32, 34, 37, 52, 60$
  $25 > 6$: si prosegue con la sottosequenza a sinistra
- secondo passo: $1, 6, 8, 10, 18$
  $8 > 6$: si prosegue con la sottosequenza a sinistra
- terzo passo: $1, 6$
  $1 < 6$: si prosegue con la sottosequenza a destra
- quarto passo: 6
  $6 = 6$: l'algoritmo termina

Risultato: l'elemento cercato è presente.

Algoritmo di ricerca binaria

Per codificare l'algoritmo di ricerca binaria si può usare un'istruzione iterativa, in ogni passo della quale si confronta l'elemento cercato con l'elemento centrale della sottosequenza in esame.

A questo scopo si possono usare tre variabili per tener traccia degli **indici** del primo e dell'ultimo elemento della sottosequenza (lista) da analizzare in ogni iterazione, e di quello in posizione centrale. Se quest'ultimo non coincide con il valore cercato, in base all'esito del confronto si modificherà il valore associato alla variabile corrispondente all'indice del primo oppure dell'ultimo elemento, in modo che nella **successiva** iterazione tali indici corrispondano agli estremi della **nuova** sottosequenza da analizzare.

Algoritmo di ricerca binaria

La funzione è disponibile nel *file* `68_ricerca_binaria.py`.
Si noti l'uso dell'operatore `//` (quoziente intero di una divisione) per ottenere l'indice dell'elemento in posizione centrale come valore intero, per tener conto di sottosequenze di lunghezza dispari.

```
def ricerca_binaria(sequenza, valore):
    inizio = 0
    fine = len(sequenza) - 1
    while inizio <= fine:
        centro = (inizio + fine)//2
        if sequenza[centro] == valore:
            return True
        if sequenza[centro] > valore:
            fine = centro - 1
        else:
            inizio = centro + 1
    return False
```

Algoritmo di ricerca binaria: efficienza

Anche per l'algoritmo di ricerca binaria il numero di confronti da eseguire dipende da S e da x.

Da questo punto di vista è evidente che il caso **migliore** è quello in cui x si trovi nella posizione **centrale** di S: in tal caso viene infatti eseguito un solo confronto.

Non è difficile convincersi che il caso **peggiore** è quello in cui x non sia presente in S, oppure si trovi in una posizione che viene analizzata solo quando la sottosequenza in esame contiene **un solo** elemento (si veda l'ultimo degli esempi precedenti): in entrambi i casi il numero di confronti sarà infatti il maggiore, per una data lunghezza della sequenza iniziale.

Algoritmo di ricerca binaria: efficienza

Il numero k di confronti che vengono eseguiti **nel caso peggiore** per una sequenza di N elementi ordinati non può essere calcolato esattamente, contrariamente alla ricerca sequenziale. Può però essere approssimato calcolando quanti confronti saranno necessari per ottenere una sottosequenza di un solo elemento, assumendo che dopo ogni confronto la lunghezza della sequenza da analizzare si riduca della metà, mediante il ragionamento schematizzato di seguito:

numero di confronti	lunghezza della sequenza	
1	N	$= N/2^0$
2	$N/2$	$= N/2^1$
3	$N/4$	$= N/2^2$
...	...	...
k	1	$= N/2^{k-1}$

Tenendo conto che k deve essere un intero, da quanto sopra si ottiene facilmente:

$$k = \lceil \log_2 N + 1 \rceil$$

Confronto tra ricerca sequenziale e binaria

È ora possibile confrontare l'efficienza dei due algoritmi di ricerca.
Il numero di confronti richiesto nel caso peggiore, per sequenze **ordinate** di N elementi, è pari a

- N, per la ricerca sequenziale
- $\lceil \log_2 N + 1 \rceil$ (circa), per la ricerca binaria

Poiché $N \geq \lceil \log_2 N + 1 \rceil$ per qualsiasi intero N, si può concludere che nel caso peggiore la ricerca binaria è più efficiente di quella sequenziale. Il vantaggio è evidente al crescere di N, come mostra il seguente esempio

	numero di confronti (caso peggiore)	
N	ricerca sequenziale: N	ricerca binaria: $\lceil \log_2 N + 1 \rceil$
10	10	4
100	100	8
1.000	1.000	11
1.000.000	1.000.000	21
10^9	10^9	31
...	...	...

Algoritmi di ordinamento

Si consideri ora il problema di **ordinare** secondo un certo criterio una data sequenza S.

Esistono diversi algoritmi di ordinamento che si differenziano per la loro efficienza. Di seguito si descrivono due algoritmi tra i più semplici, ma anche tra i meno efficienti

- ordinamento **per selezione** (*selection sort*)
- ordinamento **per inserimento** (*insertion sort*)

Altri algoritmi molto noti, alcuni dei quali più efficienti, sono *quicksort*, *heap sort* e *merge sort* (ordinamento per fusione).

Algoritmi di ordinamento

Facendo riferimento a una lista di lunghezza qualsiasi contenente numeri, l'obiettivo è **modificare** la stessa lista in modo che i suoi elementi risultino ordinati in senso crescente o decrescente.

Di seguito si considererà l'ordinamento crescente. Gli stessi algoritmi possono essere facilmente modificati per ordinare una lista in senso decrescente.

Ordinamento per selezione

L'algoritmo di **ordinamento per selezione** consiste nell'eseguire una serie di **scambi** tra **coppie** di elementi di una sequenza S in modo da disporre l'elemento più piccolo nella prima posizione, poi quello immediatamente successivo nella seconda posizione, e così via.

Detta N la lunghezza di S, questo si ottiene mediante un'iterazione sulle **posizioni** della sequenza, dalla prima alla **penultima** (la $N-1$-esima); per ogni posizione k

1. si cerca la **posizione** dell'elemento più piccolo (la si indichi con m) a partire da quello in posizione k
2. se $m \neq k$, si **scambiano** gli elementi in tali posizioni

Ordinamento per selezione: esempio

Si vuole ordinare in senso crescente la sequenza

$$52,\ -34,\ -48,\ 58,\ 75,\ -80,\ 22$$

Di seguito si mostra in rosso la posizione considerata in ogni passo dell'algoritmo e in grassetto l'elemento più piccolo da tale posizione in avanti; se i due elementi coincidono non si esegue nessuno scambio

- $52,\ -34,\ -48,\ 58,\ 75,\ \mathbf{-80},\ 22$: si scambiano 52 e $\mathbf{-80}$
- $-80,\ -34,\ \mathbf{-48},\ 58,\ 75,\ 52,\ 22$: si scambiano -34 e $\mathbf{-48}$
- $-80,\ -48,\ \mathbf{-34},\ 58,\ 75,\ 52,\ 22$: nessuno scambio
- $-80,\ -48,\ -34,\ 58,\ 75,\ 52,\ \mathbf{22}$: si scambiano 58 e $\mathbf{22}$
- $-80,\ -48,\ -34,\ 22,\ 75,\ \mathbf{52},\ 58$: si scambiano 75 e $\mathbf{52}$
- $-80,\ -48,\ -34,\ 22,\ 52,\ 75,\ \mathbf{58}$: si scambiano 75 e $\mathbf{58}$
- $-80,\ -48,\ -34,\ 22,\ 52,\ 58,\ 75$

Risultato: $\mathbf{-80},\ \mathbf{-48},\ \mathbf{-34},\ \mathbf{22},\ \mathbf{52},\ \mathbf{58},\ \mathbf{75}$.

Ordinamento per selezione

Di seguito si mostra una possibile codifica in linguaggio Python per l'ordinamento in senso crescente di una lista di numeri.

L'algoritmo è codificato sotto forma di funzione che riceve come argomento la lista da ordinare, e **modifica** la **stessa** lista disponendo i suoi elementi in ordine crescente.

Per codificare gli algoritmi di ordinamento in linguaggio Python si può sfruttare il fatto (osservato in precedenza) che, se una funzione modifica una lista ricevuta come argomento, le modifiche si riflettono sulla lista del programma chiamante. Non è quindi necessario usare l'istruzione `return` per restituire la lista modificata.

Ordinamento per selezione

L'algoritmo di ordinamento per selezione può essere codificato tramite due iterazioni nidificate.

L'iterazione principale viene svolta sugli indici della lista, dalla prima alla **penultima** posizione; l'iterazione nidificata viene eseguita sugli indici delle posizioni successive a quella considerata nell'iterazione principale, per cercare la posizione dell'elemento più piccolo.

Si noti che lo scambio di una coppia di elementi della lista (quando necessario) avviene per mezzo di una variabile ausiliaria.

Ordinamento per selezione

La funzione si trova nel *file* 69_ordinamento_per_selezione.py

```
def ordinamento_per_selezione(sequenza):
    i = 0
    while i < len(sequenza) - 1:
        indice_minimo = i
        j = i + 1
        while j < len(sequenza):
            if sequenza[j] < sequenza[indice_minimo]:
                indice_minimo = j
            j = j + 1
        if indice_minimo != i:
            temp = sequenza[i]
            sequenza[i] = sequenza[indice_minimo]
            sequenza[indice_minimo] = temp
        i = i + 1
```

Ordinamento per inserimento

L'algoritmo di **ordinamento per inserimento** opera mediante un'iterazione sugli indici della sequenza S, dalla **seconda** posizione fino all'ultima: per ogni posizione k si "estrae" da S l'elemento corrispondente e lo si **inserisce** nella posizione corretta tra i $k-1$ elementi che lo precedono.

A questo scopo si esegue un'iterazione nidificata sugli elementi in posizione $k-1, k-2, \ldots, 1$ (procedendo **a ritroso**), e si **sposta** di una posizione verso destra ogni elemento che risulti maggiore di quello nella posizione k. Se si arriva a una posizione m contenente un elemento minore o uguale a quello in posizione k, quest'ultimo viene inserito nella posizione $m+1$; se tutti i $k-1$ elementi sono già stati spostati, l'elemento in posizione k viene inserito nella prima posizione della sequenza; in entrambi i casi, il procedimento termina.

Ordinamento per inserimento: esempio

Si consideri la stessa sequenza dell'esempio precedente

$$52,\ -34,\ -48,\ 58,\ 75,\ -80,\ 22$$

Per ogni passo, in rosso si mostra la sottosequenza già ordinata, e in grassetto l'elemento da inserire in tale sottosequenza

- $52,\ \mathbf{-34},\ -48,\ 58,\ 75,\ -80,\ 22$
  l'elemento 52 viene spostato di una posizione verso destra, e -34 viene inserito nella prima posizione
- $-34,\ 52,\ \mathbf{-48},\ 58,\ 75,\ -80,\ 22$
  gli elementi -34 e 52 vengono spostati di una posizione verso destra, -48 viene inserito nella prima posizione
- $-48,\ -34,\ 52,\ \mathbf{58},\ 75,\ -80,\ 22$
  nessuno spostamento
- $-48,\ -34,\ 52,\ 58,\ \mathbf{75},\ -80,\ 22$
  nessuno spostamento

(cont.)

Esempio

(cont.)

- $-48,\ -34,\ 52,\ 58,\ 75,\ \mathbf{-80},\ 22$
  gli elementi $-48, \ldots, 75$ vengono spostati di una posizione verso destra, -80 viene inserito nella prima posizione
- $-80,\ -48,\ -34,\ 52,\ 58,\ 75,\ \mathbf{22}$
  gli elementi $52,\ 58,\ 75$ vengono spostati di una posizione verso destra, 22 viene inserito nella quarta posizione
- $-80,\ -48,\ -34,\ 22,\ 52,\ 58,\ 75$

Risultato: $\mathbf{-80},\ \mathbf{-48},\ \mathbf{-34},\ \mathbf{22},\ \mathbf{52},\ \mathbf{58},\ \mathbf{75}$.

Ordinamento per inserimento

Anche questo algoritmo può essere codificato in linguaggio Python mediante due iterazioni nidificate.

L'iterazione principale viene eseguita sugli indici della lista, dalla **seconda** all'ultima posizione.

L'iterazione nidificata viene eseguita a ritroso sugli indici delle posizioni precedenti quella considerata nell'iterazione principale, eseguendo gli opportuni confronti e spostamenti; al termine di tale iterazione l'elemento considerato nell'iterazione principale viene inserito nella posizione corretta.

Anche in questo caso è necessario usare una variabile ausiliaria per memorizzare l'elemento considerato nell'iterazione principale.

Algoritmo di ordinamento per inserimento

La funzione è disponibile nel *file*
`71_ordinamento_per_inserimento.py`

```
def ordinamento_per_inserimento(sequenza):
    i = 1
    while i < len(sequenza):
        temp = sequenza[i]
        j = i - 1
        while j >= 0 and sequenza[j] > temp:
            sequenza[j + 1] = sequenza[j]
            j = j - 1
        sequenza[j + 1] = temp
        i = i + 1
```

Algoritmi di ordinamento: efficienza

Entrambi gli algoritmi consistono in una serie di **confronti** tra coppie di valori, e in un certo numero di assegnamenti, scambi o spostamenti degli elementi della sequenza. È inoltre facile convincersi che il numero di scambi o spostamenti non può essere superiore a quello dei confronti.

L'efficienza può perciò essere valutata, in funzione della lunghezza N della sequenza, in termini del **numero di confronti** tra coppie di suoi elementi.

Si può ora osservare che nell'ordinamento per selezione il numero di confronti **non** dipende dai valori della sequenza originale. Nel caso dell'ordinamento per inserimento il numero di confronti dipende invece dalla sequenza originale, e ci si può facilmente convincere che il caso peggiore (quello che richiede il maggior numero di confronti) si verifica quando tale sequenza è ordinata nel senso **opposto** a quello desiderato.

Algoritmi di ordinamento: efficienza

L'ordinamento per selezione di una sequenza di N elementi richiede l'analisi di $N-1$ posizioni, e per ciascuna di esse l'analisi di tutte quelle successive (per trovare l'elemento più piccolo).

Il numero di confronti per ciascuna delle $N-1$ posizioni considerate è quindi il seguente

- posizione 1: $N-1$ confronti
- posizione 2: $N-2$ confronti
- ...
- posizione $N-1$: **un** confronto

Il numero totale di confronti è quindi:

$$(N-1)+(N-2)+\ldots+1 \quad = \quad \sum_{k=1}^{N-1} k \quad = \quad \frac{N(N-1)}{2}$$

Con un ragionamento analogo si può ricavare che nel caso peggiore l'ordinamento per inserimento richiede lo **stesso** numero di confronti.

Algoritmi di ordinamento: efficienza

Si può concludere che, per una sequenza di N elementi, il numero di operazioni richieste nel caso peggiore dai due algoritmi di ordinamento considerati è **proporzionale** a N^2.

Tra gli altri algoritmi citati in precedenza, nel caso peggiore

- *quicksort* ha la stessa efficienza di *selection sort* e *insertion sort*
- *merge sort* e *heap sort* richiedono $N \log_2 N$ confronti

Algoritmi di ricerca e ordinamento: esercizi

1. Modificare la funzione Python per l'algoritmo di ricerca binaria in modo che sia in grado di elaborare una lista di numeri ordinati in senso **decrescente**
2. Modificare le funzioni Python per gli algoritmi di ordinamento per selezione e per inserimento in modo che siano in grado ordinare in senso **decrescente** una lista di numeri
3. Modificare le funzioni Python per gli algoritmi di ricerca e ordinamento in modo che siano in grado di elaborare valori diversi da numeri; alcuni esempi
   - liste di caratteri, per esempio: `["c", "m", "a", "r"]`
   - liste di date memorizzate in dizionari, per esempio: `{"g": 1, "m": 2, "a": 2020}`

Lettura e scrittura di dati su *file*

Meccanismi di *input*/*output*

L'unico meccanismo di acquisizione dei dati d'ingresso (*input*) da parte di un programma Python visto finora è basato sulla funzione `input`, che consente di acquisire sequenze di caratteri scritte attraverso la tastiera (mentre il programma è in esecuzione) nella *shell*.

Similmente, per comunicare all'utente i risultati delle elaborazioni svolte da un programma (*output*) si è usata la funzione `print`, che consente di stampare nella *shell* il valore di qualsiasi espressione.

Oltre a queste modalità di *input*/*output* ne esistono altre, per esempio l'uso di interfacce grafiche e sistemi di puntamento (*mouse*, ecc.), e l'uso dei *file* della memoria secondaria.

Gestione dei *file* nel linguaggio Python

I *file* della memoria secondaria possono essere usati come strumento per

- acquisire dati precedentemente memorizzati in un *file*, per esempio da un utente o da un altro programma (*input*)
- memorizzare in un *file* i risultati prodotti da un programma (*output*)

Dal punto di vista di un programma Python un *file* consiste in una **sequenza ordinata** di valori. Più precisamente, un *file* può essere

- una sequenza di **byte**, i cui valori possono essere visti come numeri naturali nell'insieme $\{0, 1, \ldots, 255\}$
- una sequenza di **caratteri** (*file* di **testo**)

In questo corso si considerano solo i *file* di testo.

Gestione dei *file* nel linguaggio Python

Nel linguaggio Python l'accesso ai *file* è reso possibile attraverso diverse funzioni predefinite, le quali consentono di eseguire su un *file* di testo due tipi di operazioni

- **lettura**: acquisizione di una **sequenza** di caratteri del *file*, sotto forma di una **stringa**
- **scrittura**: memorizzazione nel *file* di una sequenza di caratteri contenuti in una **stringa**

Come si vedrà, per la lettura e scrittura su *file* sono di grande utilità le funzioni predefinite per l'elaborazione di stringhe viste in precedenza: `str`, `split`, `strip`, `int` e `float`.

Procedimento per l'accesso ai *file*

La lettura o la scrittura di dati su un *file* avvengono in tre fasi

1. **apertura** del *file*, per mezzo della funzione predefinita `open`
2. esecuzione di una o più operazioni di lettura oppure di scrittura, per mezzo di funzioni predefinite
3. **chiusura** del *file*, mediante la funzione predefinita `close`

Procedimento per l'accesso ai *file*

La funzione `open` consente di indicare il nome del *file* al quale si vuole accedere e la **modalità** di accesso desiderata (lettura o scrittura): tale operazione viene detta **apertura** del *file*.

Da un *file* aperto in modalità di **lettura** è possibile solo **acquisire** dati; in un *file* aperto in modalità di **scrittura** è invece possibile solo **memorizzare** dati. Il tentativo di memorizzare dati in un *file* aperto in modalità di lettura, o di acquisire dati da un *file* aperto in modalità di scrittura, produce un errore.

La funzione `close` **chiude** il *file*, cioè impedisce di eseguire ulteriori operazioni su di esso (fino a che non venga eventualmente riaperto con un'altra chiamata di `open`).

Apertura di un *file*: la funzione `open`

La funzione `open` restituisce un valore di un tipo specifico (`_io.TextIOWrapper`) contenente alcune informazioni sul *file*. Il valore restituito deve essere **assegnato** a una variabile, che dovrà essere usata in ogni successiva operazione sullo stesso *file*, inclusa la chiusura. Per gli scopi di questo corso non sono necessari ulteriori dettagli sul valore restituito da `open`.

Sintassi: variabile = `open(`nome-*file*`,` modalità`)`

- variabile: il nome della variabile che verrà associata al *file*
- nome-*file*: una **stringa** contenente il nome del *file*
- modalità: una **stringa** che indica la modalità di apertura (lettura o scrittura)

Apertura di un *file*: la funzione `open`

Il nome del *file* che si desidera aprire deve essere passato come argomento della funzione `open` sotto forma di **stringa**.

Il nome del *file* può essere

- **assoluto**, cioè preceduto dalla sequenza (***pathname***) dei nomi delle *directory* che lo contengono, a partire dalla *directory* **radice** del *file system*, secondo la sintassi prevista dal sistema operativo del proprio calcolatore
- **relativo**, cioè composto dal solo nome del *file*, se il *file* si trova nella **stessa** *directory* del programma

Apertura di un *file*: la funzione `open`

Per esempio, nel caso di un *file* di nome `dati.txt` che si trovi nella *directory* `C:\Users\Dati\` di un sistema operativo Windows

- il nome **relativo** è `dati.txt`
- il nome **assoluto** è `C:\Users\Dati\dati.txt`

Se un *file* con lo stesso nome (`dati.txt`) è memorizzato nella *directory* `/Users/Dati/` di un sistema operativo Linux oppure Macintosh

- il nome **relativo** è ancora `dati.txt`
- il nome **assoluto** è `/Users/Dati/dati.txt`

Modalità di accesso ai *file*

È possibile aprire in modalità di lettura solo un *file* **esistente**. Se il *file* non esiste si otterrà un errore.

La modalità di scrittura consente invece anche la **creazione** di un nuovo *file*. Più precisamente, attraverso la modalità di scrittura è possibile

- **creare** un **nuovo** *file*
- **aggiungere** dati **al termine** di un *file* già **esistente**
- **sovrascrivere** (cancellare e sostituire) il contenuto di un *file* già **esistente**

Modalità di accesso ai *file*

Nella chiamata di `open` la modalità di accesso è indicata (come secondo argomento) da una **stringa** composta da un singolo carattere

- `"r"` (*read*): **lettura** (se il *file* non esiste si ottiene un **errore**)
- `"w"` (*write*): (sovra)**scrittura**
  - se il *file* **non** esiste viene **creato**
  - se il *file* **esiste** viene **sovrascritto**, cancellando i dati che contiene
- `"a"` (*append*): **scrittura** (aggiunta)
  - se il *file* **non** esiste viene **creato**
  - se il *file* **esiste** i nuovi dati saranno **aggiunti** a quelli già presenti, in coda al *file*

Apertura di un *file* in lettura: esempi

Si assuma che la *directory* `C:\Users\user\Dati\` di un sistema operativo Windows contenga un *file* di testo di nome `dati.txt` (creato da un programma Python come si vedrà più avanti, o con un qualsiasi altro programma, per esempio un *editor* di testi come `Blocco note`).

Per aprire tale *file* in modalità di **lettura**, memorizzando il valore restituito da `open` in una variabile di nome `f`, si dovrà scrivere l'istruzione

```
f = open("C:/Users/user/Dati/dati.txt", "r")
```

Si noti che, anche per i sistemi operativi Windows, nella stringa contenente il nome assoluto di un *file* il linguaggio Python consente di usare come separatore il carattere `/` invece del carattere `\`.
Questo consente di evitare ambiguità nel caso di nomi di *file* o *directory* che iniziano con il carattere `n`, dato che nella stringa corrispondente comparirebbe la sequenza `"\n"` che Python interpreta come *newline*.
In alternativa si può scrivere:

```
f = open("C:\\Users\\user\\Dati\\dati.txt", "r")
```

Apertura di un *file* in lettura: esempi

Nel caso in cui l'istruzione mostrata in precedenza si trovi in un programma memorizzato nella **stessa** *directory* nella quale si trova il *file* da aprire (in questo esempio, `C:\Users\user\Dati\`), sarà anche possibile usare il nome **relativo** del *file*:

```
f = open("dati.txt", "r")
```

Apertura di un *file* in lettura: esempi

Si ricordi che

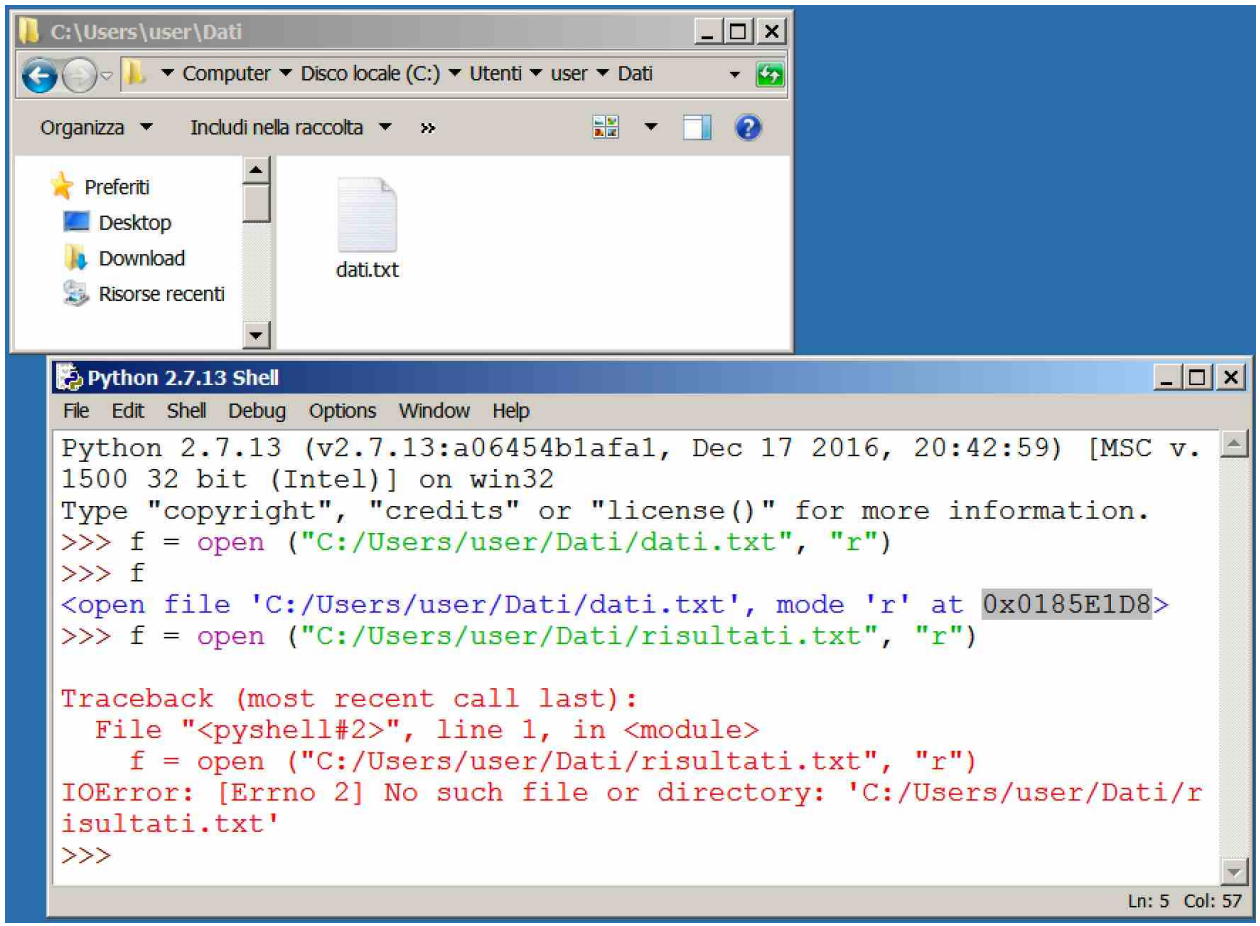
- tentare di aprire in **lettura** un *file* inesistente produce un errore
- se la chiamata di `open` è scritta nella *shell* bisogna indicare il nome **assoluto** del *file* (in caso contrario l'interprete cercherà il *file* in una delle *directory* di installazione dell'ambiente Python)

Come per qualsiasi valore Python, è possibile visualizzare nella *shell* una rappresentazione del valore restituito da una chiamata di *open*, per esempio scrivendo come espressione il nome della variabile a cui tale valore è stato assegnato. Il risultato che si ottiene è simile al seguente:

```
<_io.TextIOWrapper name='C:/Users/user/dati.txt'
mode='r' encoding='cp1252'>
```

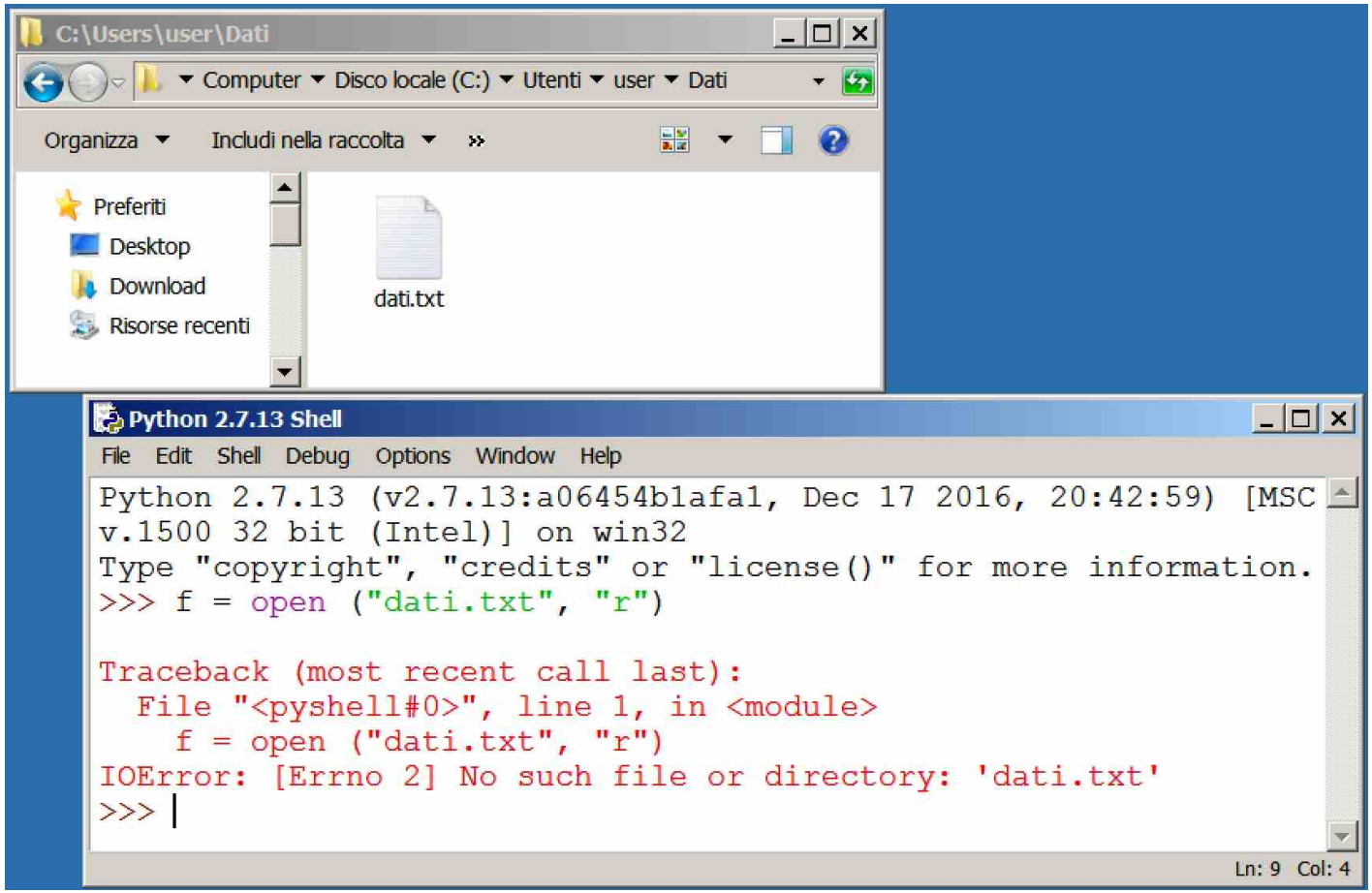
Apertura di un *file* in lettura: esempi

Nella *directory* `C:\Users\user\Dati\` è presente solo il *file* `dati.txt`. Le istruzioni scritte nella *shell* aprono in lettura tale *file* (indicandone il nome assoluto), e tentano di aprire un *file* inesistente (`risultati.txt`).



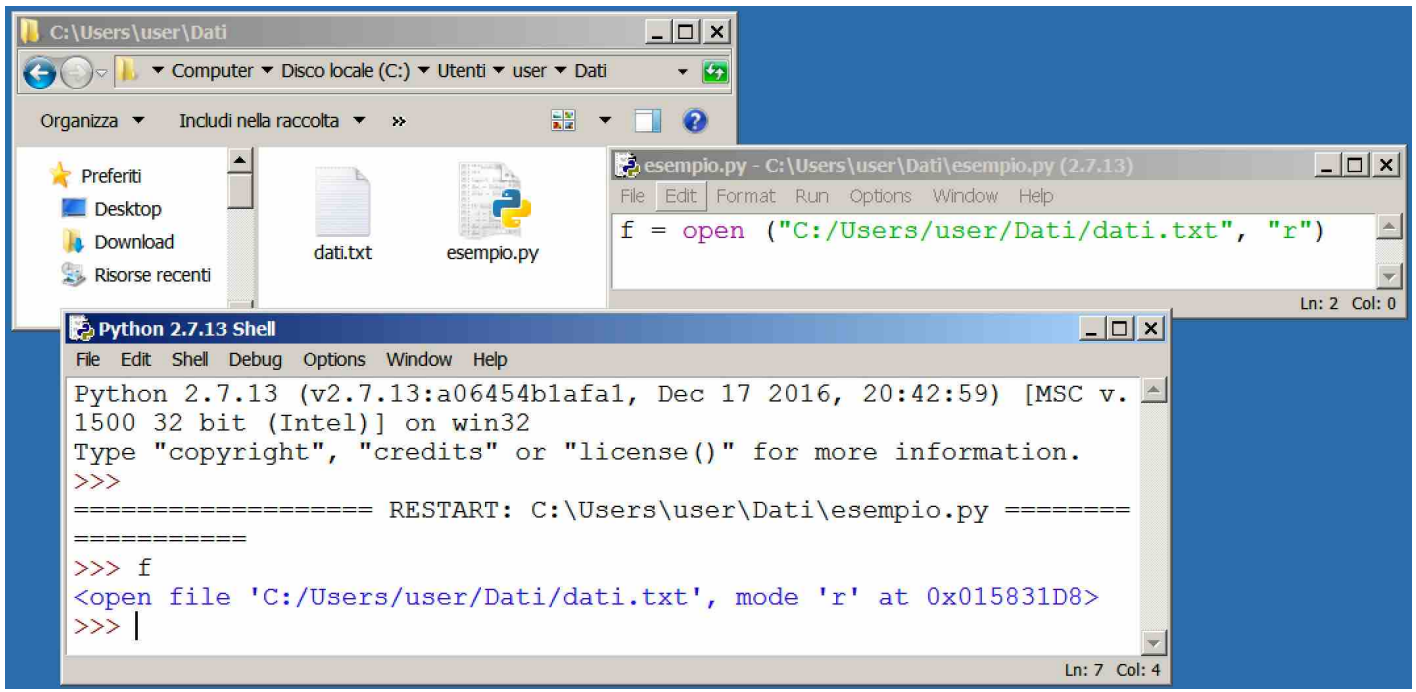
Apertura di un *file* in lettura: esempi

L'istruzione scritta nella *shell* tenta di aprire il *file* `dati.txt` nella *directory* `C:\Users\user\Dati\` indicandone erroneamente il nome relativo



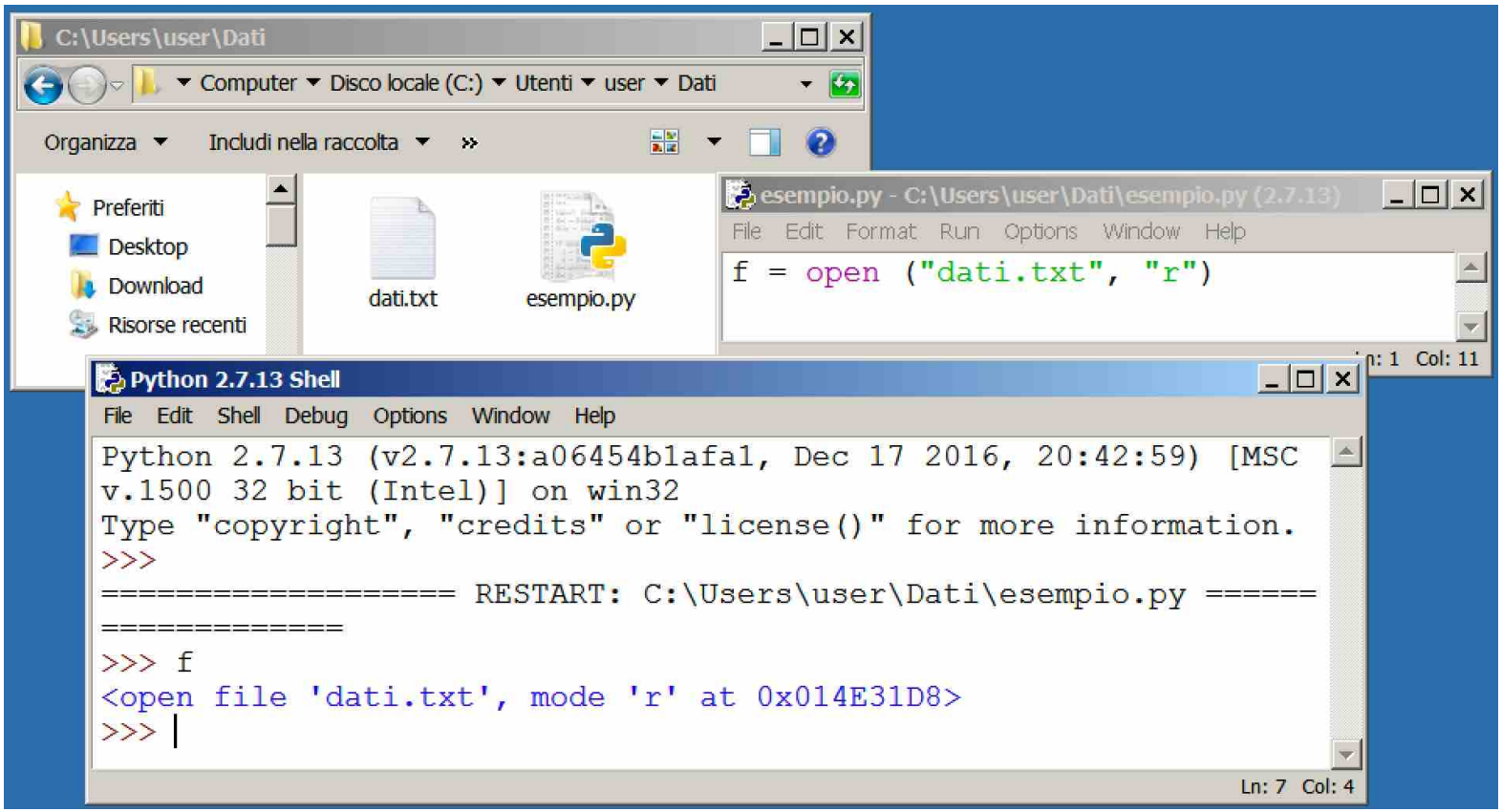
Apertura di un *file* in lettura: esempi

Il programma nel *file* `esempio.py`, all'interno della *directory* `C:\Users\user\Dati\`, può aprire in lettura il *file* `dati.txt`, memorizzato nella stessa *directory*, indicandone il nome assoluto...



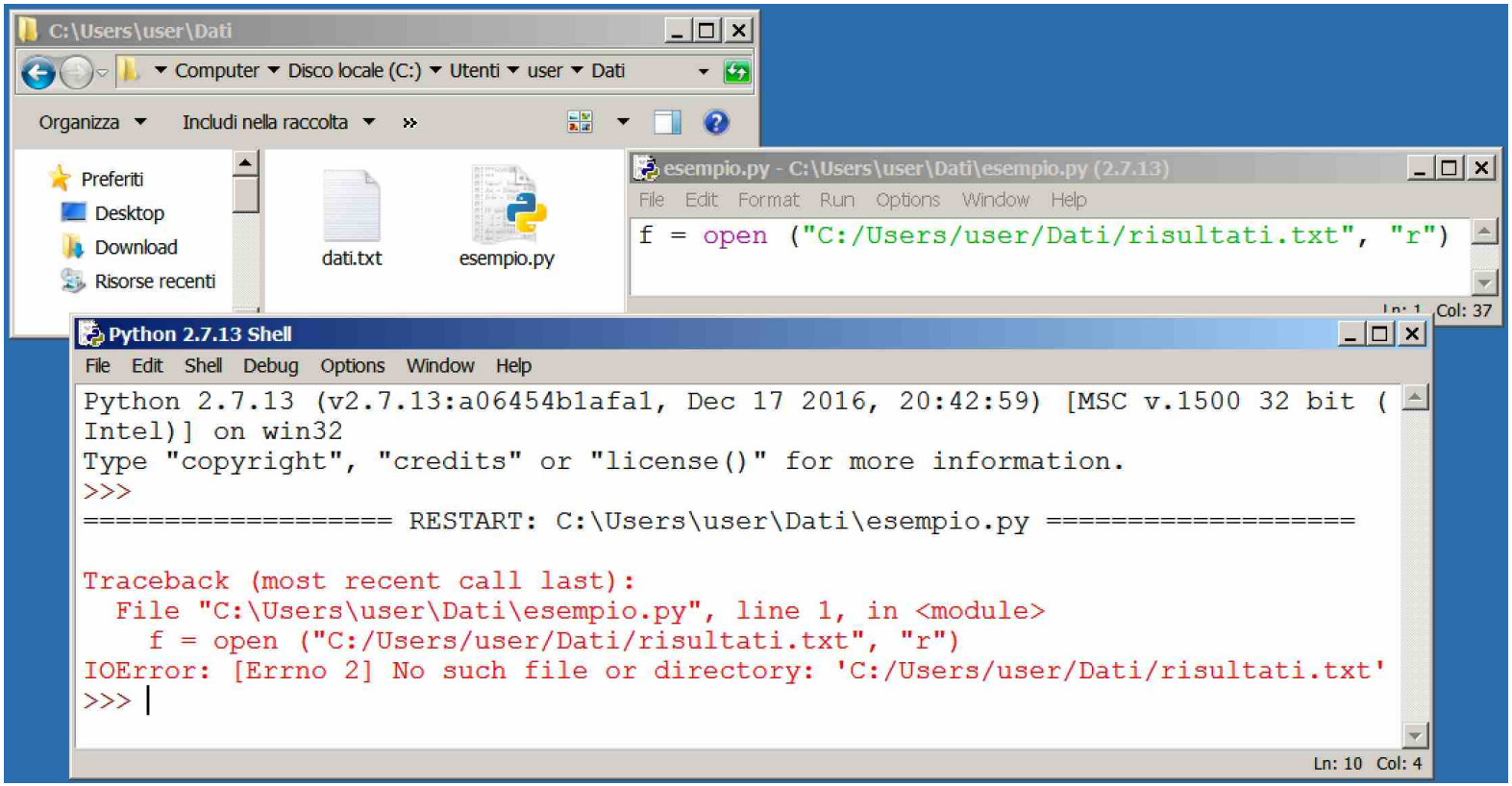
Apertura di un *file* in lettura: esempi

...oppure il nome relativo



Apertura di un *file* in lettura: esempi

Tentativo di apertura in lettura di un *file* inesistente (`risultati.txt`), da parte di un programma



Apertura di un *file* in scrittura: esempi

Si consideri ancora la *directory* `C:\Users\user\Dati\`
Per creare un **nuovo** *file* di nome `dati2.txt` al suo interno e associarlo a una variabile di nome `nf` si potrà usare una qualunque delle due modalità di scrittura:

```
nf = open("C:/Users/user/Dati/dati2.txt", "w")
```

oppure

```
nf = open("C:/Users/user/Dati/dati2.txt", "a")
```

Anche in questo caso è possibile usare il nome relativo del *file*, purché la chiamata di `open` **non** sia scritta nella *shell*, **e** il programma che la contiene si trovi nella **stessa** *directory* del *file*:

```
nf = open("dati2.txt", "w")
```

oppure

```
nf = open("dati2.txt", "a")
```

Apertura di un *file* in scrittura: esempi

Se si volesse aprire in scrittura un *file* già esistente (come il *file* `dati.txt` nella *directory* `C:\Users\user\Dati\` dell'esempio precedente), **cancellando** gli eventuali dati in esso presenti, si dovrà usare la modalità `"w"` (in questo esempio si assume di voler associare il *file* a una variabile di nome `fw`):

```
fw = open("C:/Users/user/Dati/dati.txt", "w")
```

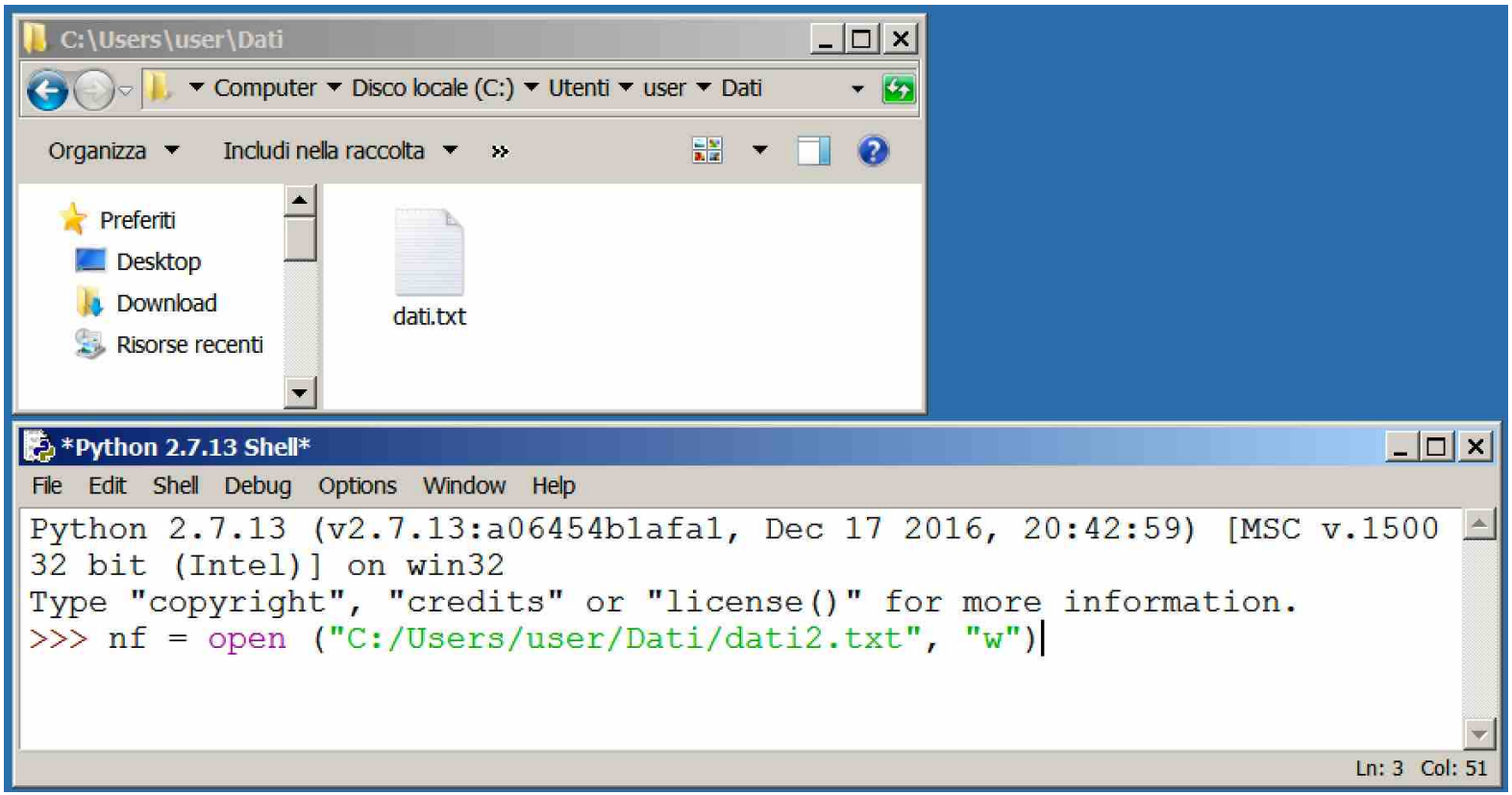
Se invece si volesse aprire in scrittura un *file* già esistente per **aggiungere** dati a quelli eventualmente già presenti (senza cancellare questi ultimi) si dovrà usare la modalità `"a"`. Con riferimento allo stesso esempio di sopra:

```
fw = open("C:/Users/user/Dati/dati.txt", "a")
```

Anche in questo caso è possibile usare il nome relativo del *file*, nelle condizioni indicate in precedenza.

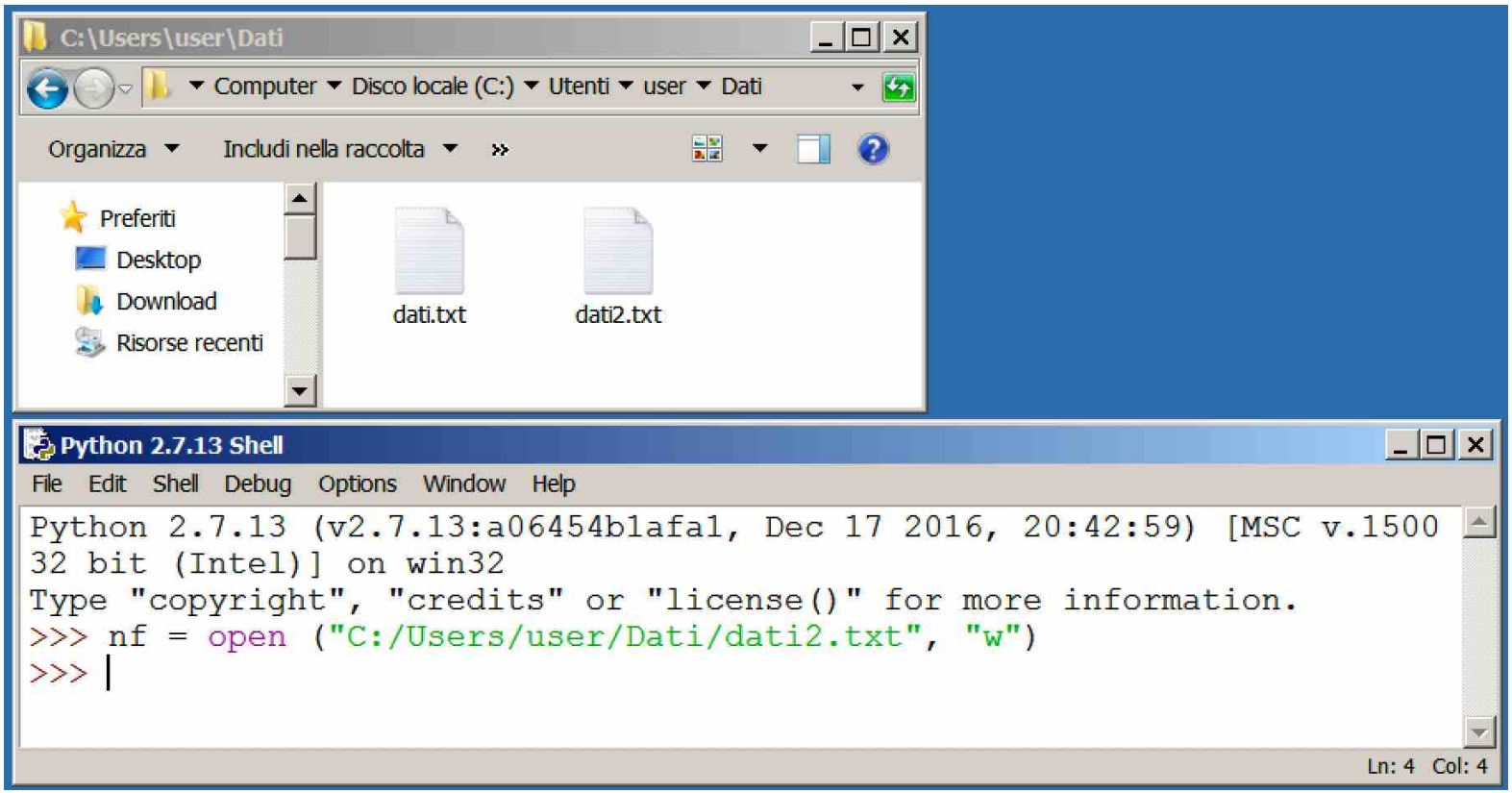
Apertura di un *file* in scrittura: esempi

Creazione di un nuovo *file* (`dati2.txt`) e apertura in scrittura, in modalità `"w"` (dalla *shell*): **prima** della chiamata di `open`...



Apertura di un *file* in scrittura: esempi

...e **dopo** la chiamata



Chiusura di un *file*: la funzione `close`

Quando le operazioni di lettura o scrittura su un *file* sono terminate, il *file* deve essere chiuso attraverso la funzione predefinita `close`. Questo impedirà l'esecuzione di ulteriori operazioni su tale *file*, fino a che esso non venga eventualmente riaperto.

Sintassi: variabile`.close()`

dove variabile deve essere la variabile usata nell'apertura dello **stesso** *file* attraverso la funzione `open` (si noti la sintassi particolare della chiamata di `close`, analoga a quella di funzioni come `split`).

Per esempio, assumendo che un *file* di nome `dati.txt` sia stato aperto in lettura con l'istruzione

```
f = open("dati.txt", "r")
```

esso dovrà essere chiuso mediante la chiamata

```
f.close()
```

Apertura contemporanea di più *file*

Se lo si desidera è anche possibile aprire più file contemporaneamente. A questo scopo nel momento dell'apertura è necessario associare ogni *file* a una **diversa** variabile.

Per esempio, se si volesse aprire in lettura un *file* esistente di nome `dati.txt`, e creare un nuovo *file* di nome `risultati.txt`, associandoli rispettivamente alle variabili `f_dati` e `f_risultati`, le operazioni di apertura e chiusura sarebbero le seguenti:

```
f_dati = open("dati.txt", "r")
f_risultati = open("risultati.txt", "w")
...
f_dati.close()
f_risultati.close()
```

Scrittura in un *file*: la funzione `write`

In un *file* di testo che sia stato aperto in scrittura (in modalità `"w"` oppure `"a"`) è possibile scrivere dati sotto forma di **stringhe** (cioè sequenze di caratteri) attraverso la funzione predefinita `write`.

Sintassi: variabile`.write(`stringa`)`

- variabile è la variabile associata al *file*
- stringa è una **stringa** contenente la sequenza di caratteri da scrivere nel *file*

Si ricordi che non è possibile scrivere dati in un *file* aperto in **lettura** (in modalità `"r"`): una chiamata di `write` su un tale *file* produrrebbe un errore.

Scrittura in un *file*: la funzione `write`

Mentre un *file* è aperto in scrittura la funzione `write` può essere chiamata più volte per scrivere nel *file* diverse sequenze di caratteri.

Qualsiasi chiamata di `write` scrive la corrispondente sequenza di caratteri **al termine** del *file*, cioè dopo gli eventuali caratteri già presenti al suo interno.

Scrittura in un *file*: esempi

L'esecuzione di un programma contenente le istruzioni

```
f = open("testo.txt", "w")
f.write("Prima riga.\nSeconda riga.")
f.close()
```

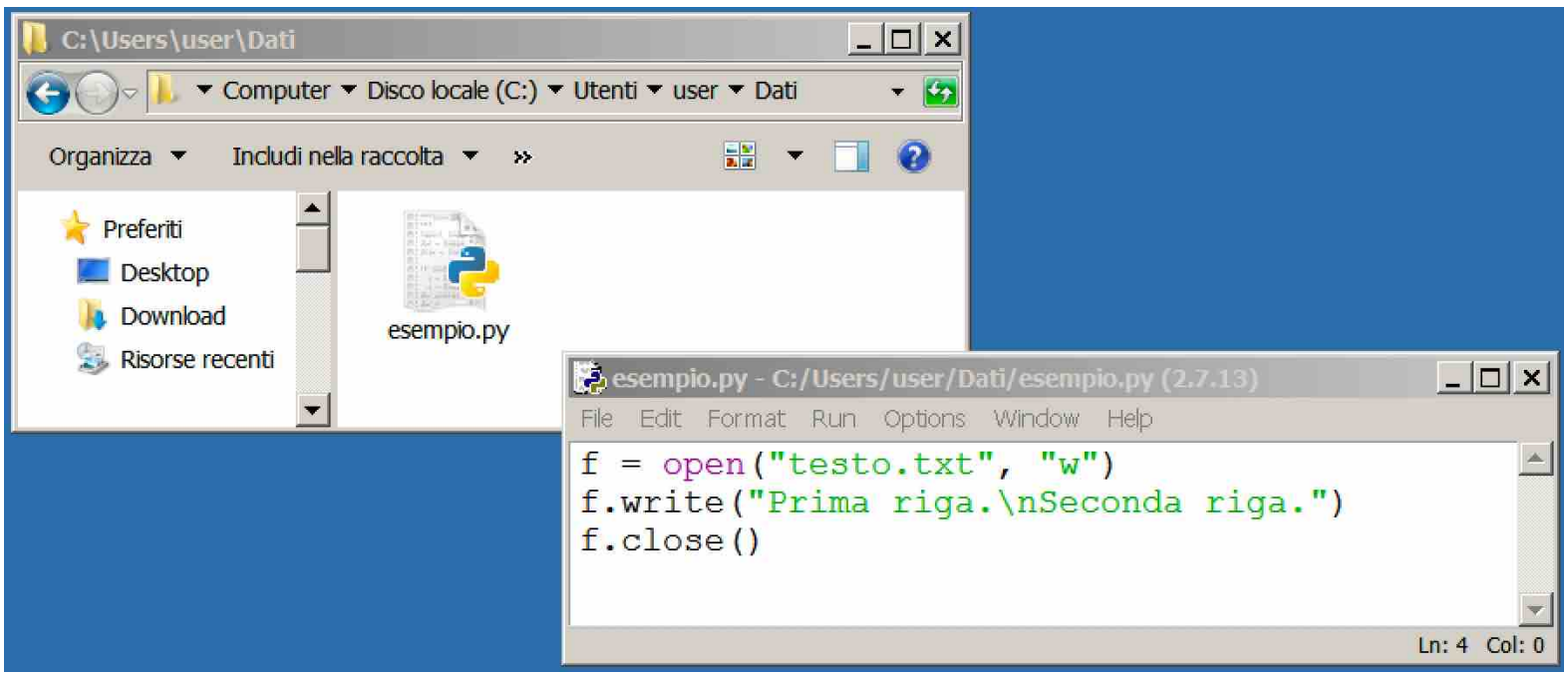
crea un *file* di nome `testo.txt` nella stessa *directory* del programma (se il *file* già esiste, ne cancella il contenuto) e scrive al suo interno la stringa indicata.

Aprendo successivamente il *file* con un qualsiasi *editor* di testi (come `Blocco note`) si osserverà il seguente contenuto (si noti l'interruzione di riga in corrispondenza del carattere *newline*):

```
Prima riga.
Seconda riga.
```

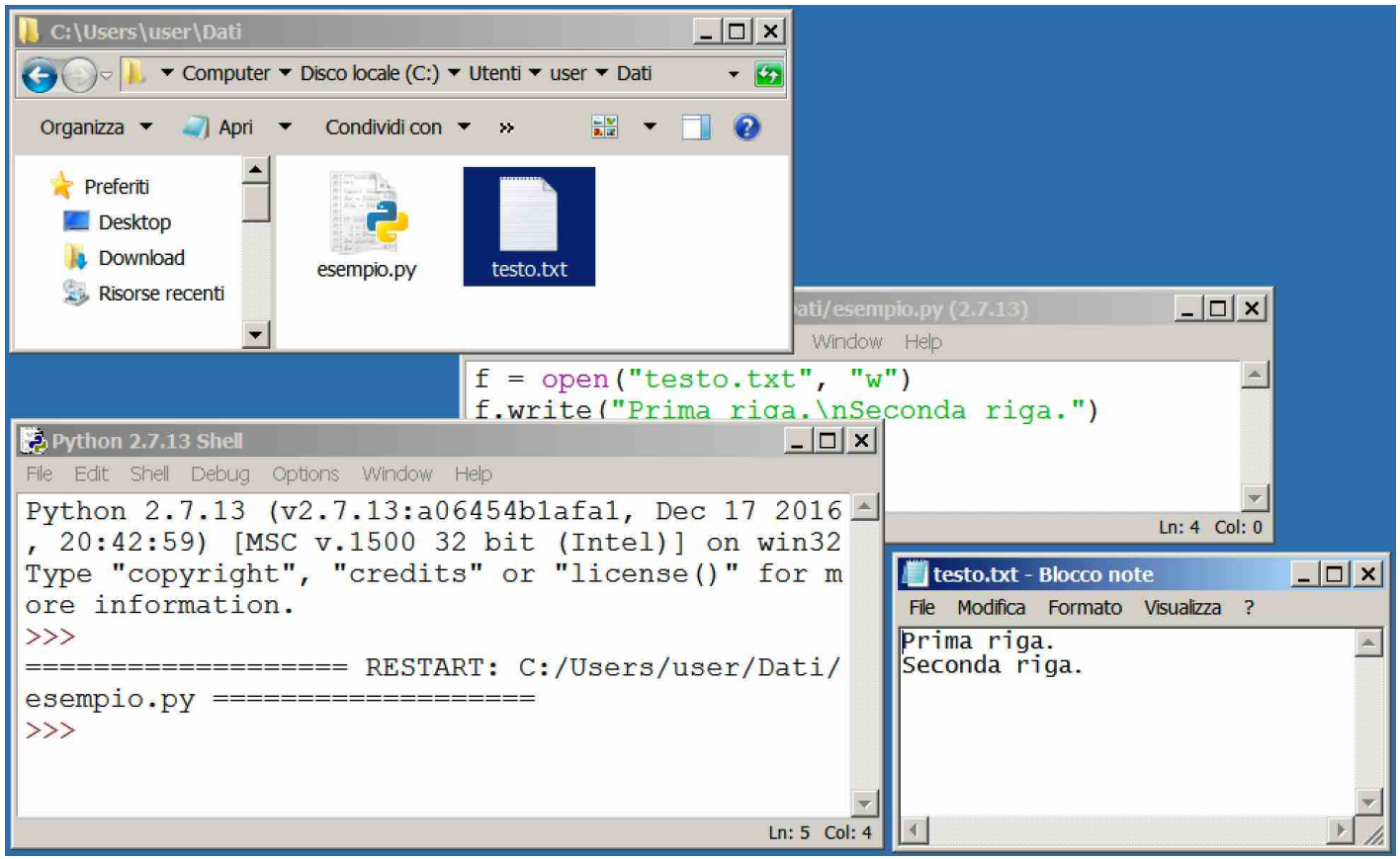
Scrittura in un *file*: esempi

Un programma che crea un nuovo *file* (`testo.txt`) nella stessa *directory* (usando il nome relativo) e lo apre in scrittura (modalità `"w"`): **prima** dell'esecuzione del programma...



Scrittura in un *file*: esempi

...e **dopo** l'esecuzione e la visualizzazione del contenuto del *file* con il programma `Blocco note`



Scrittura in un *file*: esempi

Se si riapre il *file* in scrittura, in modalità `"a"`, sarà possibile **aggiungere** del testo **al termine** dello stesso *file*, senza cancellare il testo esistente.

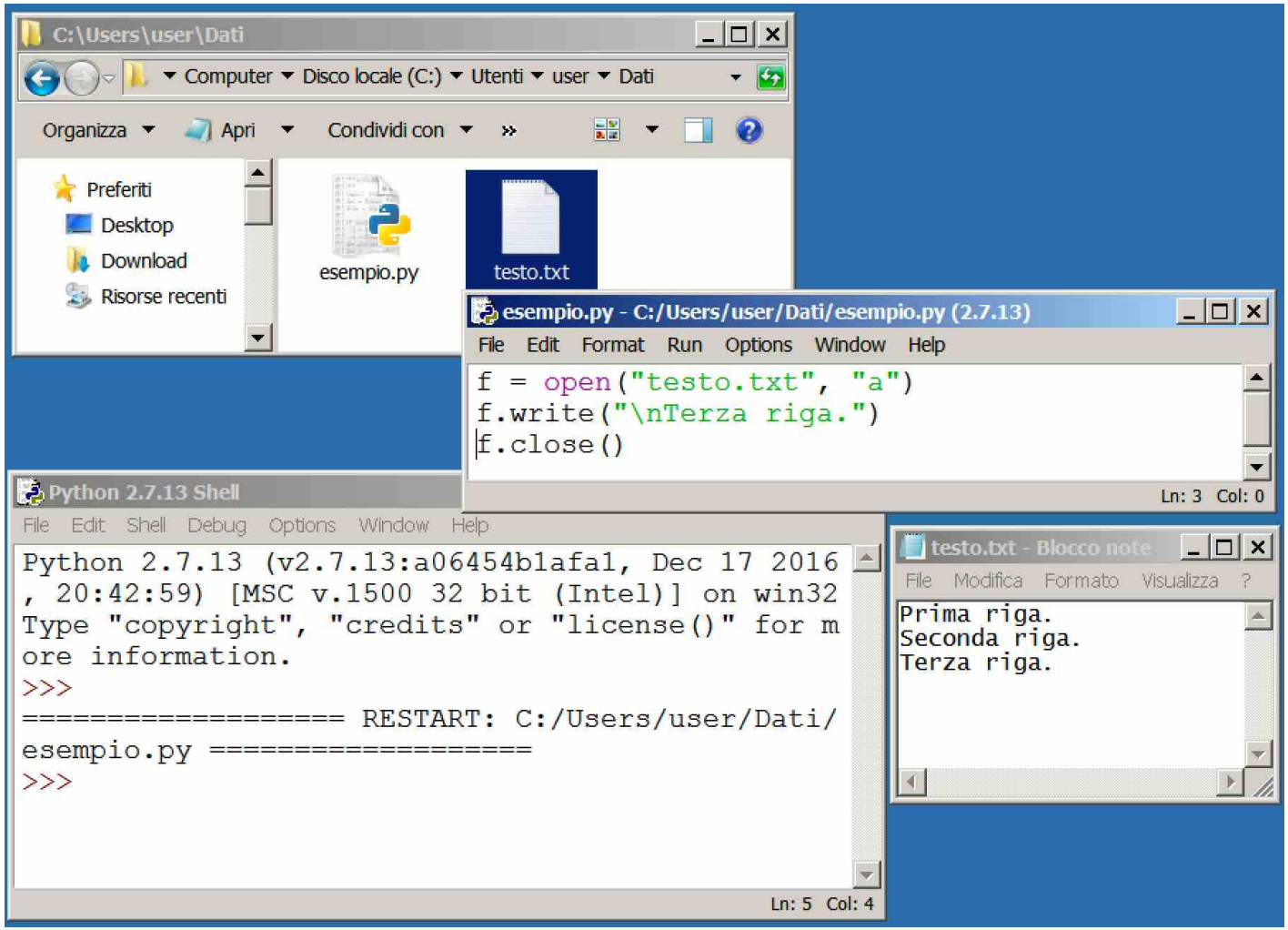
Per esempio, dopo l'esecuzione del seguente programma, anch'esso memorizzato nella stessa *directory* del *file* `testo.txt`:

```
f = open("testo.txt", "a")
f.write("\nTerza riga.")
f.close()
```

il contenuto del *file* sarà:

```
Prima riga.
Seconda riga.
Terza riga.
```

Scrittura in un *file*: esempi



Scrittura in un *file*: esempi

Se successivamente si riaprisse il *file* in scrittura, in modalità `"w"`, il suo contenuto verrebbe cancellato.

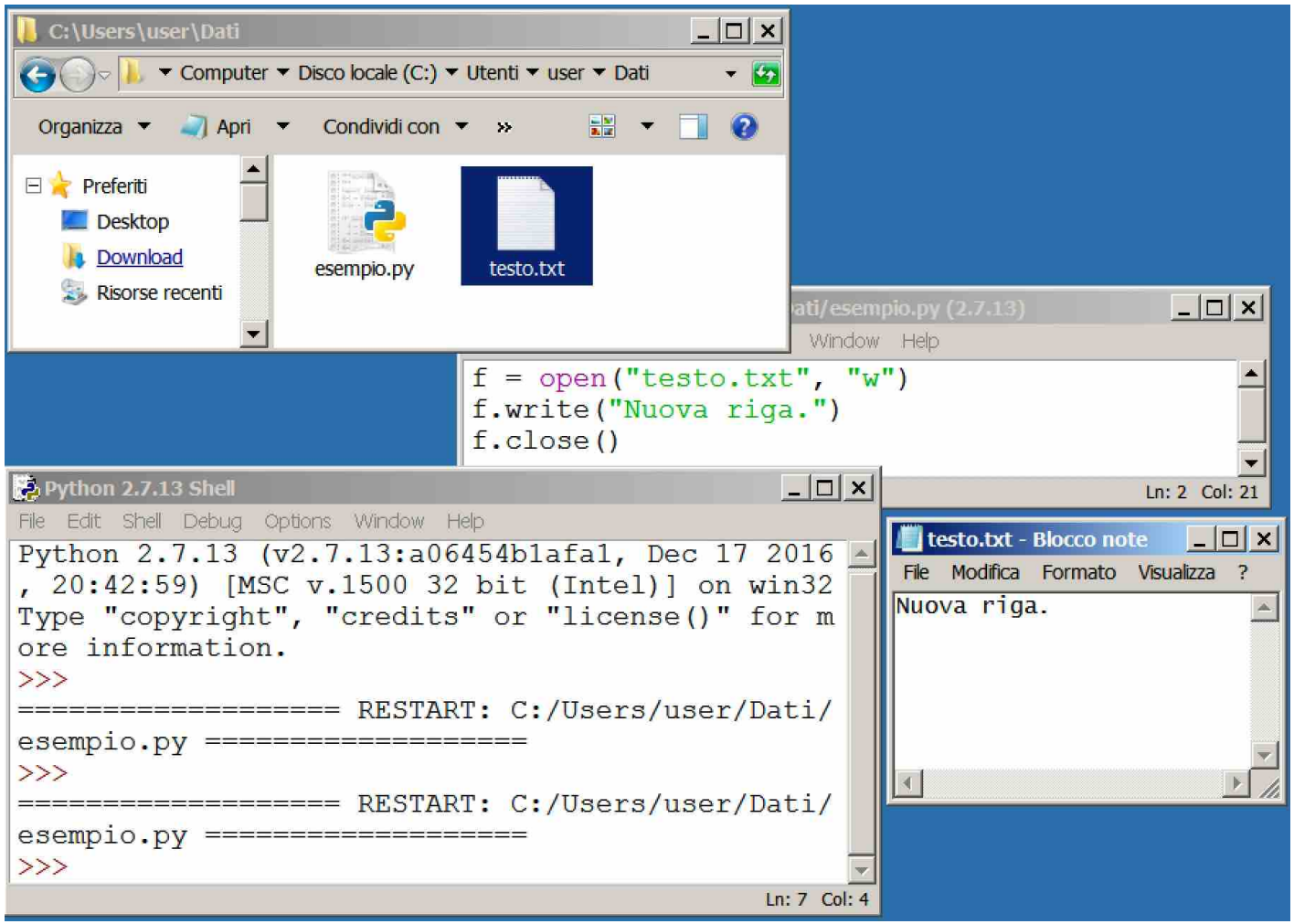
Per esempio, se si memorizzasse nella stessa *directory* il seguente programma e lo si eseguisse

```
f = open("testo.txt", "w")
f.write("Nuova riga.")
f.close()
```

il contenuto del *file* diventerebbe:

```
Nuova riga.
```

Scrittura in un *file*: esempi



Lettura da un *file*

Le operazioni di **lettura** da un *file* possono essere eseguite attraverso tre diverse funzioni predefinite

- `read`
- `readline`
- `readlines`

Queste funzioni restituiscono una parte del contenuto del *file*, o l'intero contenuto, sotto forma di **una stringa** oppure di **una lista di stringhe**.

Il valore da esse restituito deve di norma essere associato a una variabile per poter essere successivamente elaborato.

Lettura da un *file*: la funzione `read`

La funzione `read` acquisisce **l'intero contenuto** di un *file*, restituendolo sotto forma di **una stringa**. Le eventuali interruzioni di riga presenti nel *file* vengono codificate mediante il carattere *newline* `"\n"`.

La **sintassi** della chiamata di `read` (così come quella di `readline` e `readlines`) è analoga a quella della funzione `close`:

```
variabile.read()
```

dove variabile è la variabile che era stata associata al *file* al momento della sua apertura.

La funzione `read`: esempio

Si consideri un *file* di nome `testo.txt` (creato per esempio con il programma `Blocco note`):

```
Prima riga.
Seconda riga.
Terza riga.
```

Si memorizzi ora il seguente programma nella **stessa** *directory* di tale *file*, e lo si esegua:

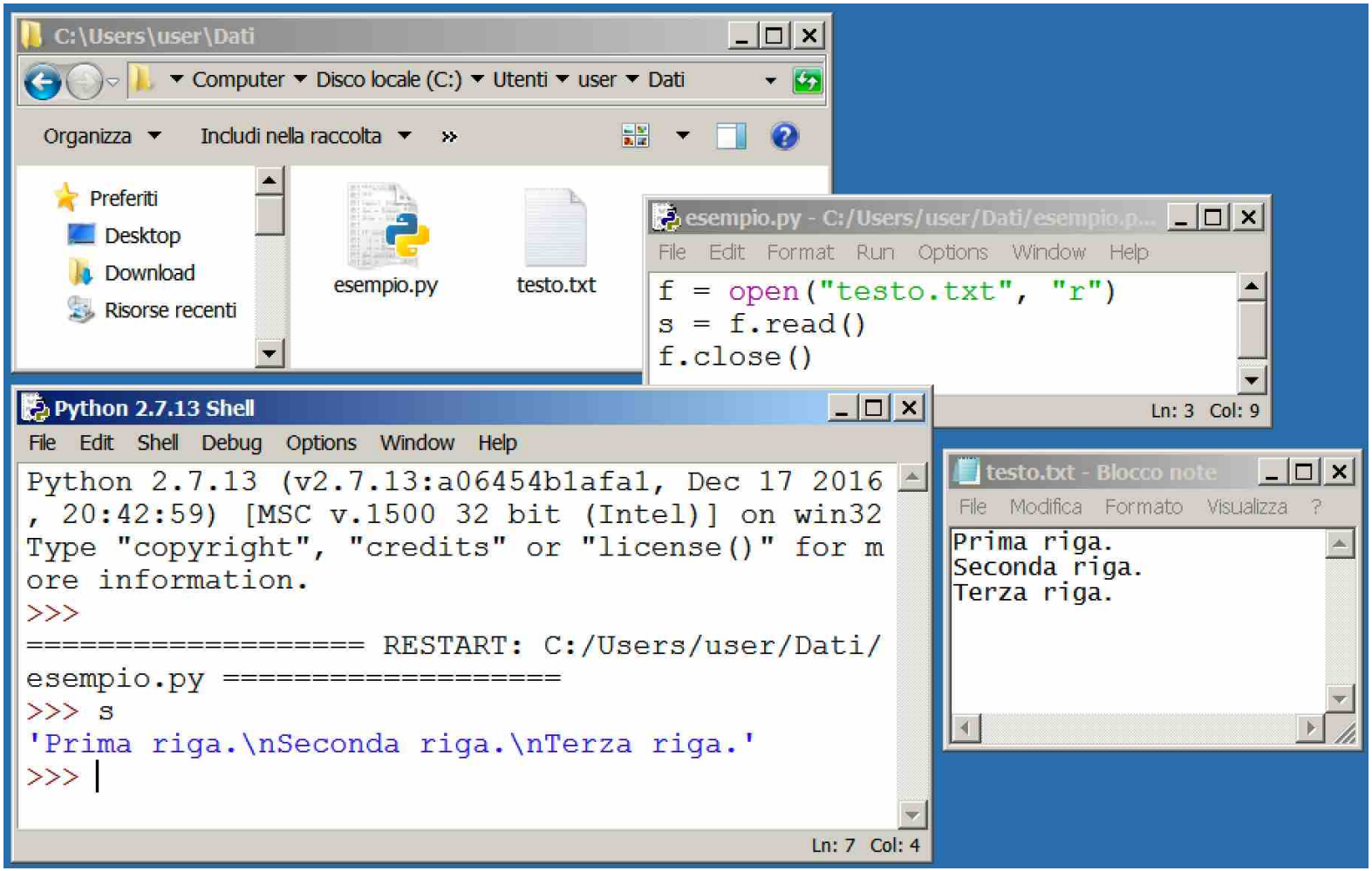
```
f = open("testo.txt", "r")
s = f.read()
f.close()
```

La variabile `s` sarà associata alla stringa seguente, come si potrà osservare valutando l'espressione `s` nella *shell*:

```
"Prima riga.\nSeconda riga.\nTerza riga."
```

Si noti la codifica delle interruzioni di riga con il carattere *newline*.

La funzione `read`: esempio



Lettura da un *file*: la funzione `read`

La **prima** chiamata di `read` dopo l'apertura (in modalità di lettura) di un *file* ne acquisisce l'intero contenuto.

Se si eseguono due o più chiamate di `read` mentre un *file* è aperto, ogni chiamata successiva alla prima restituirà **una stringa vuota**, poiché l'intero contenuto del *file* è già stato acquisito.

La funzione `read`: esempi

Riprendendo l'esempio precedente, si consideri lo stesso *file* di testo e si modifichi il programma come segue

```
f = open("testo.txt", "r")
s1 = f.read()
f.close()
f = open("testo.txt", "r")
s2 = f.read()
f.close()
```

Dopo l'esecuzione del programma, il valore associato a entrambe le variabili `s1` e `s2` sarà la stringa

```
"Prima riga.\nSeconda riga.\nTerza riga."
```

La funzione `read`: esempi

Si modifichi ora il programma come segue

```
f = open("testo.txt", "r")
s1 = f.read()
s2 = f.read()
s3 = f.read()
f.close()
```

Dopo l'esecuzione del programma, `s1` sarà associata alla stringa:

```
"Prima riga.\nSeconda riga.\nTerza riga."
```

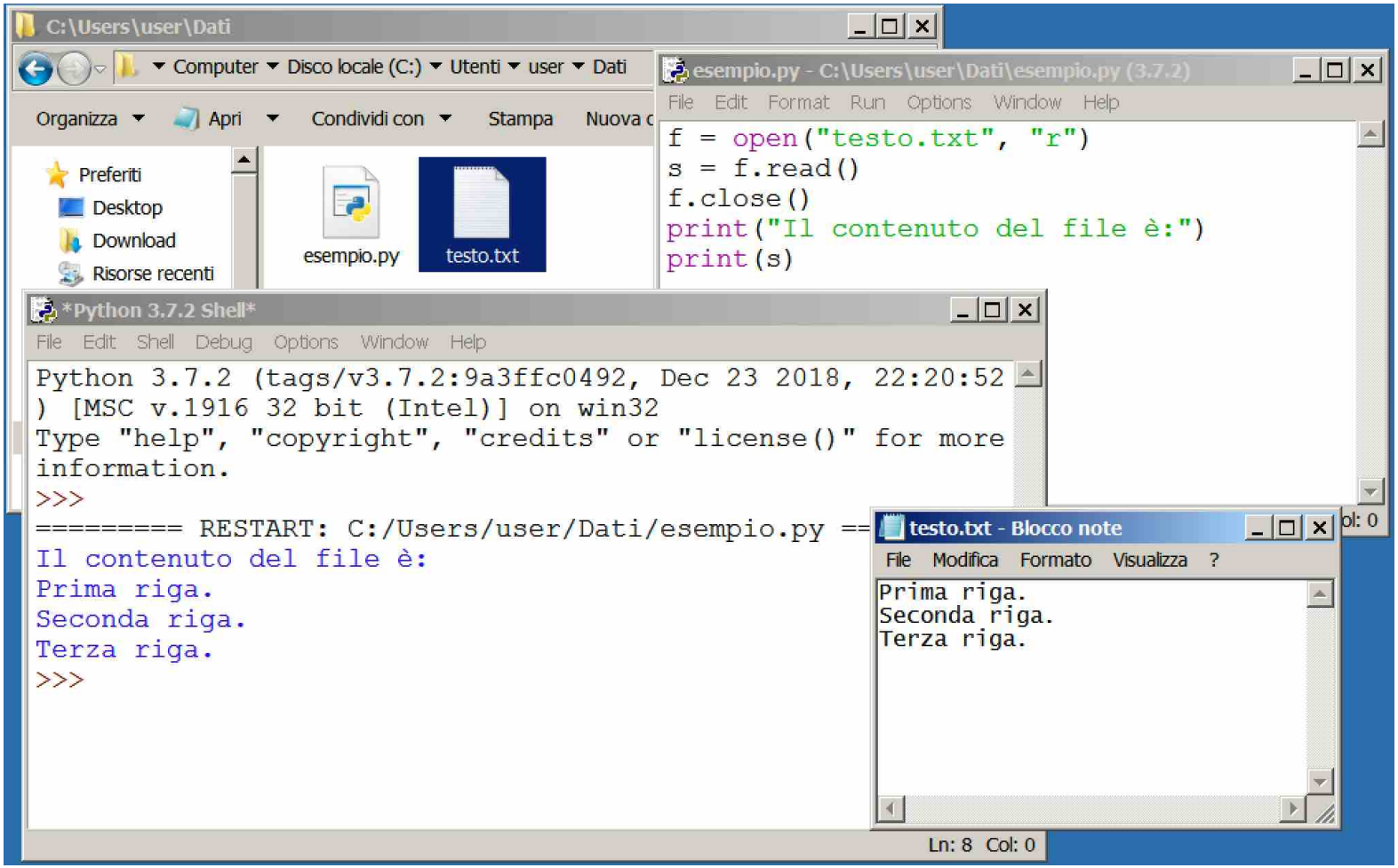
mentre `s2` e `s3` saranno associate a una stringa vuota.

La funzione `read`: esempi

Il seguente programma acquisisce il contenuto di un *file* di nome `testo.txt` (che deve trovarsi nella stessa *directory*) e lo stampa nella *shell*, inserendo un'interruzione di riga in corrispondenza di ogni eventuale carattere *newline*

```
f = open("testo.txt", "r")
s = f.read()
f.close()
print("Il contenuto del file è:")
print(s)
```

La funzione `read`: esempi



Lettura da un *file*: la funzione `readline`

La funzione `readline` acquisisce **una singola riga** di un *file*, restituendola sotto forma di **una stringa**.

Per "riga" s'intende una sequenza di caratteri fino alla prima interruzione di riga, se presente, altrimenti l'intero contenuto del *file*.

Nel primo caso anche l'interruzione di riga, codificata con il carattere *newline*, farà parte della stringa restituita da `readline`: per eliminarlo si può usare la funzione `strip`.

La **sintassi** è la seguente

```
variabile.readline()
```

dove variabile indica come al solito la variabile associata al *file*.

Lettura da un *file*: la funzione `readline`

La **prima** chiamata di `readline` dopo l'apertura (in lettura) di un *file* acquisisce la **prima** riga.

Ogni eventuale altra chiamata di `readline` prima che il *file* venga chiuso acquisisce la riga **successiva** a quella acquisita dalla chiamata precedente.

Dopo che tutte le righe di un *file* siano state acquisite da una sequenza di chiamate di `readline`, ogni chiamata successiva restituirà **una stringa vuota**.

La funzione `readline`: esempio

Si consideri lo stesso *file* `testo.txt` degli esempi precedenti (contenente tre righe), e si esegua il seguente programma dopo averlo memorizzato nella stessa *directory* del *file*

```
f = open("testo.txt", "r")
a = f.readline()
b = f.readline()
c = f.readline()
d = f.readline()
f.close()
```

Le variabili `a`, `b`, `c` e `d` saranno associate rispettivamente alle stringhe

`"Prima riga.\n"`
`"Seconda riga.\n"`
`"Terza riga."`
`""` (una stringa vuota)

Lettura da un *file*: la funzione `readline`

La funzione `readline` è utile quando si desidera acquisire ed elaborare una singola riga alla volta di un dato *file*.

Questo si può ottenere attraverso una sequenza di chiamate di `readline` all'interno di un'istruzione iterativa, che dovrà terminare non appena `readline` restituirà una stringa vuota (si ricordi che ciò indica che l'intero contenuto del *file* è già stato acquisito).

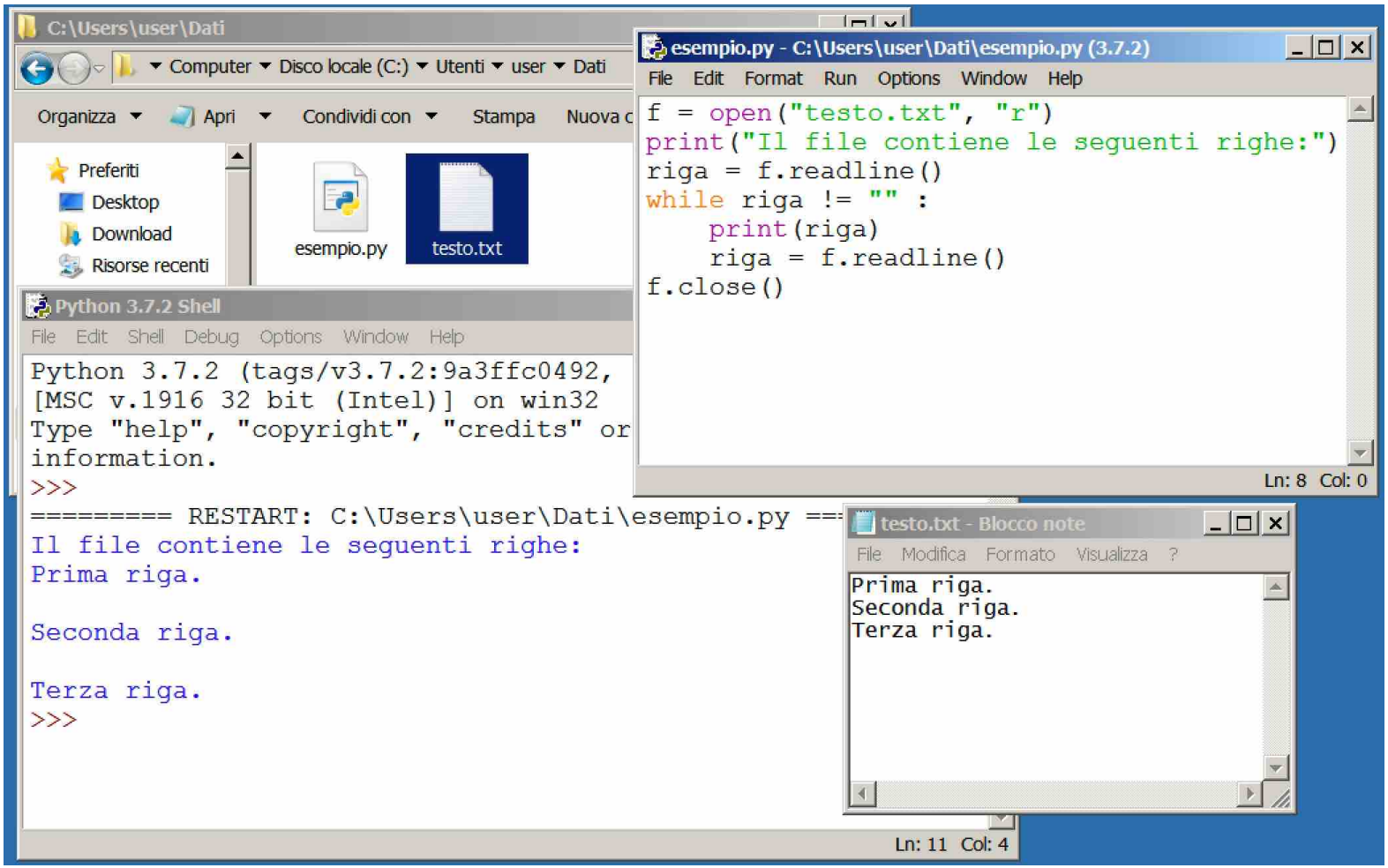
Un semplice esempio di tale procedimento è mostrato di seguito.

La funzione `readline`: esempi

Il programma seguente acquisisce e stampa nella *shell* ogni singola riga di un *file* di nome `testo.txt` (che deve trovarsi nella stessa *directory* del programma)

```
f = open("testo.txt", "r")
print("Il file contiene le seguenti righe:")
riga = f.readline()
while riga != "".
    print(riga)
    riga = f.readline()
f.close()
```

La funzione `readline`: esempi



La funzione `readline`: esempi

Si noti che, se un *file* contiene righe "vuote" **seguite** da altre righe, le prime contengono in realtà un'interruzione di riga, cioè il carattere *newline*, e quindi **non** sono realmente vuote.

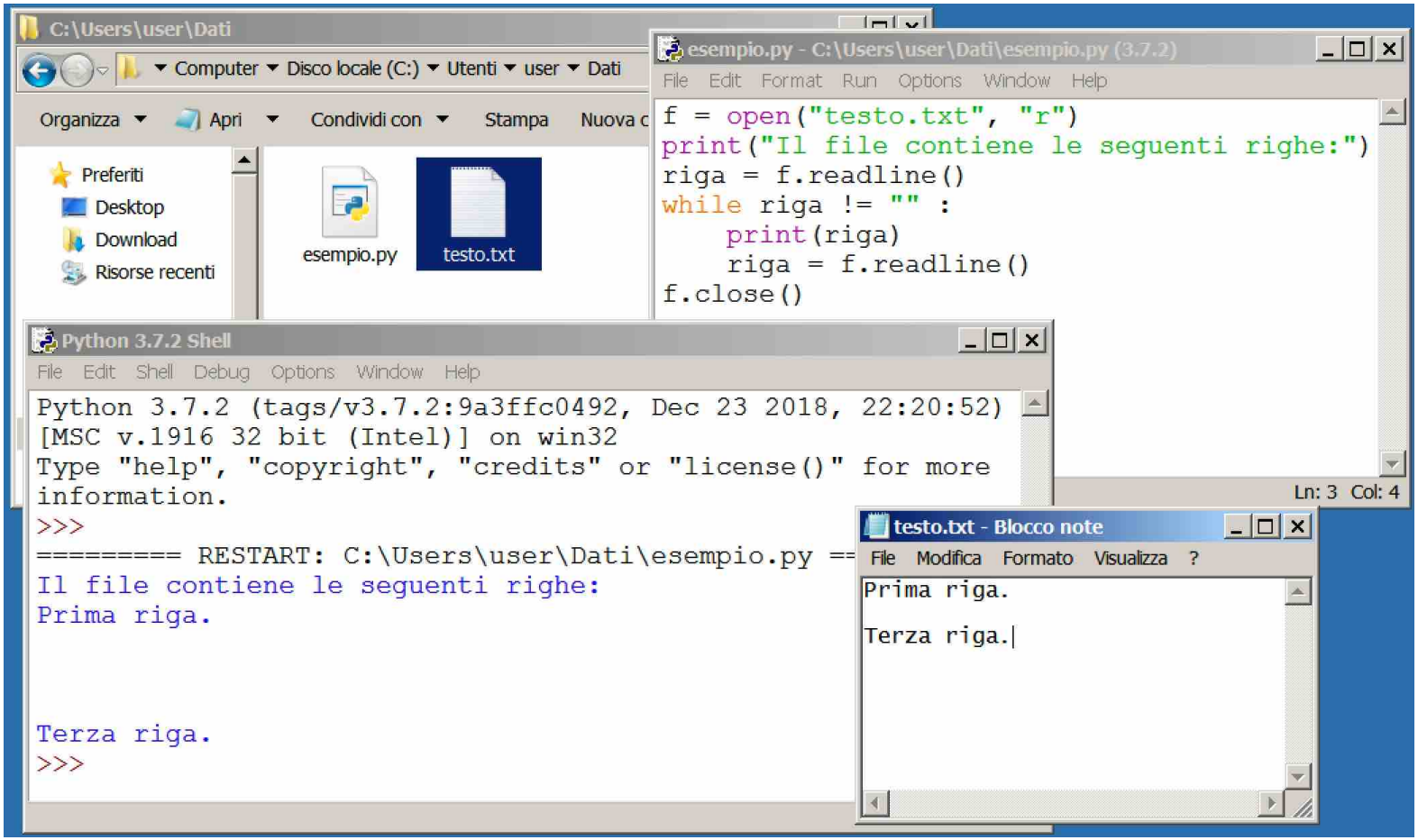
Per questo motivo l'iterazione del programma precedente termina **non** quando s'incontra la prima riga "vuota", ma quando si è raggiunta l'effettiva fine del *file*.

Per rendersi conto di questo, si provi ad eseguire lo stesso programma su un *file* come il seguente, contenente tre righe, di cui la seconda "vuota"

```
Prima riga.

Terza riga.
```

La funzione `readline`: esempi



Lettura da un *file*: la funzione `readline`

Una chiamata di `read` dopo una o più chiamate di `readline` acquisirebbe il contenuto del *file* a partire dalla riga **successiva** all'ultima acquisita mediante `readline`, fino al termine del *file*.

In altre parole, per ogni operazione di lettura eseguita con qualsiasi funzione, valgono le seguenti regole generali

- l'acquisizione inizia dal punto in cui si era conclusa la precedente operazione di lettura
- se l'intero contenuto del *file* è già stato acquisito dalle operazioni precedenti, viene restituita una stringa vuota

Lettura da un *file*: la funzione `readlines`

La funzione `readlines` acquisisce l'**intera** sequenza di caratteri contenuta in un *file*, e la restituisce sotto forma una **lista di stringhe**, ciascuna delle quali contiene **una riga** del *file*, incluso l'eventuale carattere *newline* (che potrà essere eliminato, se lo si desidera, applicando la funzione `strip` su **ciascuna** stringa).

La **sintassi** è la seguente:

```
variabile.readlines()
```

dove variabile è ovviamente la variabile associata al *file*.

Anche `readlines` è utile quando si desidera elaborare **separatamente** ogni riga di un *file*, ma, a differenza di `readline`, essa consente l'acquisizione dell'intero *file* con una sola chiamata.

La funzione `readlines`: esempio

Si consideri ancora un *file* di nome `testo.txt` il cui contenuto sia:

```
Prima riga.
Seconda riga.
Terza riga.
```

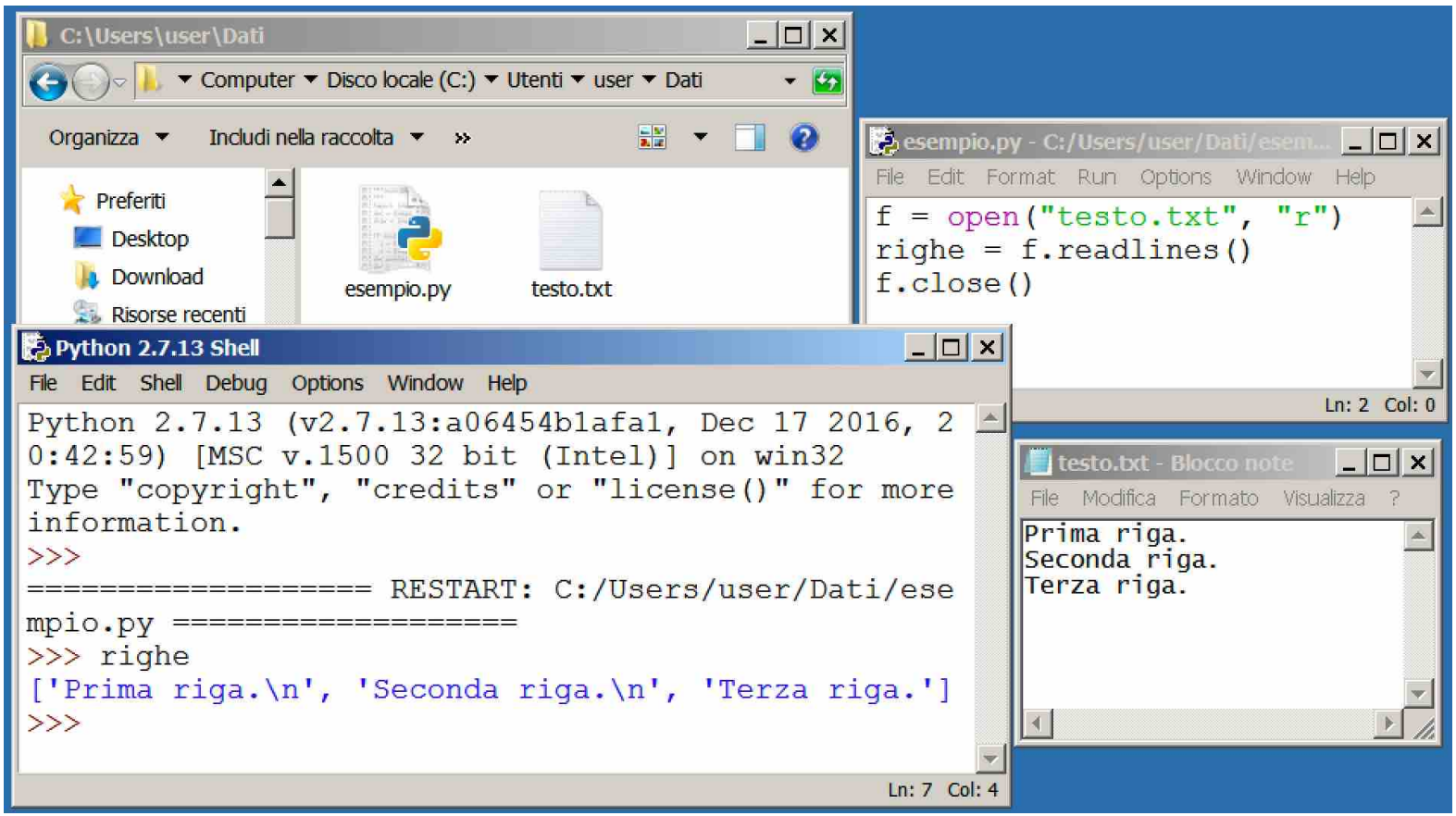
Si esegua ora il seguente programma, dopo averlo memorizzato nella stessa *directory* che contiene il *file*

```
f = open("testo.txt", "r")
righe = f.readlines()
f.close()
```

Alla variabile `righe` sarà associata la lista

```
["Prima riga.\n", "Seconda riga.\n", "Terza riga."]
```

La funzione `readlines`: esempio



Lettura da un *file*: la funzione `readlines`

Il comportamento della funzione `readlines` è simile a quello di `read`

- la **prima** chiamata di `readlines` dopo l'apertura (in modalità di lettura) di un *file* ne acquisisce l'**intero** contenuto
- se si eseguono due o più chiamate di `readlines` mentre un *file* è aperto, ogni chiamata **successiva alla prima** restituirà **una lista vuota**, poiché l'intero contenuto del *file* è già stato acquisito

La funzione `readlines`: esempi

Si modifichi il programma dell'esempio precedente come indicato di seguito

```
f = open("testo.txt", "r")
righe1 = f.readlines()
f.close()
f = open("testo.txt", "r")
righe2 = f.readlines()
righe3 = f.readlines()
f.close()
```

Dopo l'esecuzione di tale programma alle variabili `righe1` e `righe2` sarà associata la lista

```
["Prima riga.\n", "Seconda riga.\n", "Terza riga."]
```

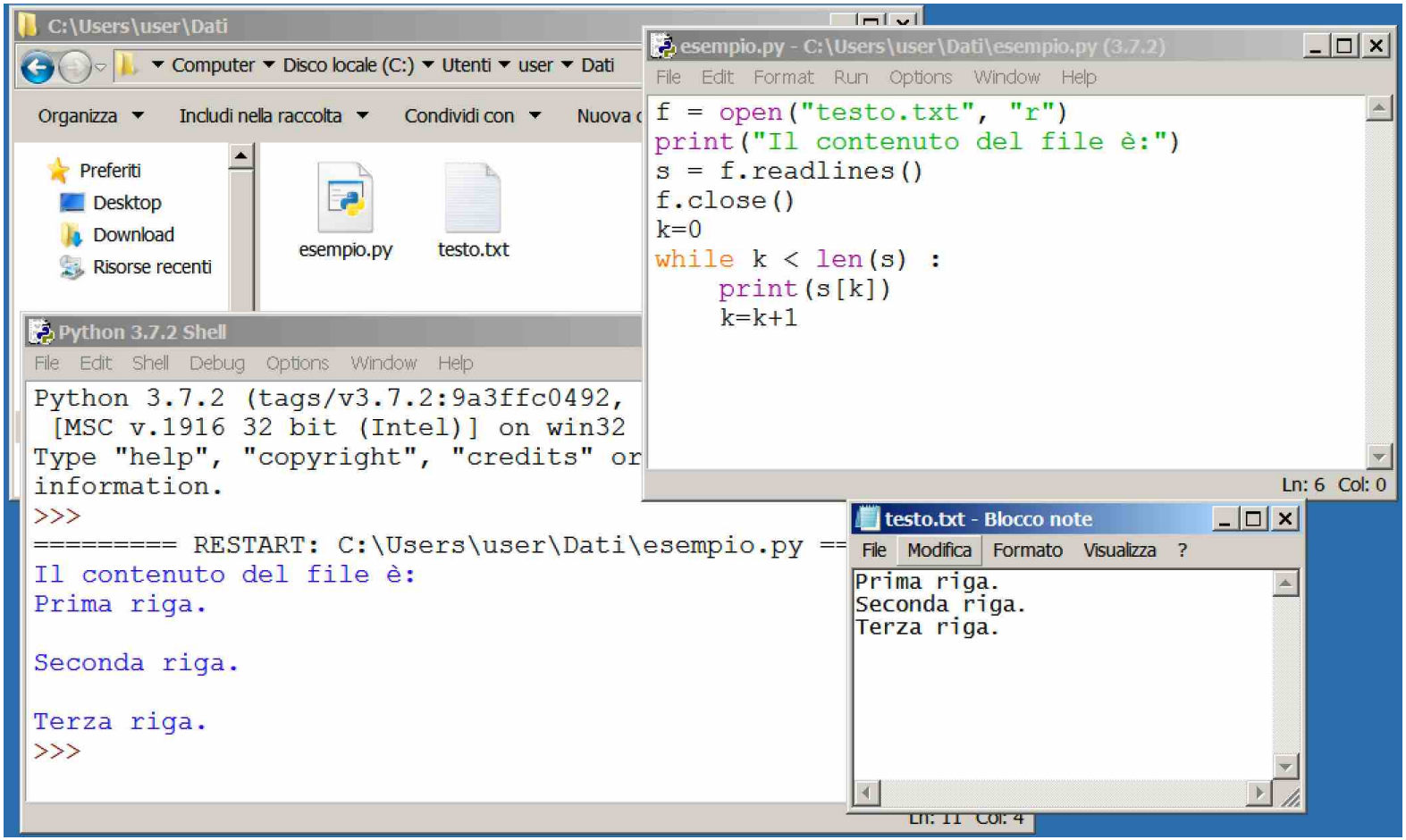
Alla variabile `righe3` sarà invece associata una lista vuota.

La funzione `readlines`: esempi

Il programma seguente è un semplice esempio che mostra come acquisire il contenuto di un *file* (di nome `testo.txt`) usando la funzione `readlines`, ed elaborare separatamente ciascuna riga mediante un'iterazione sulla lista restituita da tale funzione (in questo caso ogni riga viene stampata nella *shell*)

```
f = open("testo.txt", "r")
print("Il contenuto del file è:")
s = f.readlines()
f.close()
k = 0
while k < len(s):
    print(s[k])
    k = k + 1
```

La funzione `readlines`: esempi



La funzione `readlines`: esempi

Lo stesso programma dell'esempio precedente può essere riscritto usando l'istruzione `for` al posto di `while`

```
f = open("testo.txt", "r")
print("Il contenuto del file è:")
righe = f.readlines()
f.close()
for riga in righe:
    print(riga)
```

Lettura da un *file*: `read`, `readline`, `readlines`

Le tre funzioni viste finora sono **equivalenti** dal punto di vista dell'acquisizione di dati da un *file* di testo. In altre parole, qualsiasi operazione di lettura si possa svolgere con una di esse si potrà svolgere anche con le altre due.

Tuttavia una data operazione potrebbe richiedere programmi di diversa **complessità** a seconda della funzione che si intende usare. È quindi importante individuare la funzione più appropriata da usare in un dato contesto.

Per esempio, se s'intende elaborare separatamente ciascuna riga di un *file* è consigliabile usare `readline` o `readlines`, dato che entrambe suddividono automaticamente le righe del *file*; usando `read` sarà invece il programmatore a dover scrivere ulteriori istruzioni per individuare il termine di ciascuna riga, in corrispondenza del carattere *newline*.

Lettura da un *file*: `read`, `readline`, `readlines`

Alcune linee guida per la scelta tra le tre funzioni

- `read` è conveniente se **non** si devono acquisire ed elaborare separatamente le righe del *file*, e si vuole associare l'intero contenuto a una variabile
- `readline` è invece la scelta più opportuna se si vuole acquisire ed elaborare separatamente **alcune** righe del *file*, oppure tutte le righe, ma si vuole evitare (per esempio, per ridurre l'occupazione di memoria) di associare l'intero contenuto del *file* a una variabile
- `readlines` è preferibile se si vuole acquisire ed elaborare separatamente ciascuna riga del *file*, e si vuole associare l'intero contenuto del *file* a una variabile (una lista di stringhe)

Gli esempi mostrati di seguito comprendono i diversi casi d'uso sopra descritti.

Lettura/scrittura su *file*: esercizi

Si vuole scrivere un programma che chiede all'utente di inserire nella *shell* un testo composto da una o più righe (ciascuna conclusa dalla pressione del taso `INVIO`), terminando con l'inserimento di una riga vuota (cioè con la pressione del tasto `INVIO` **all'inizio** di una riga). Il testo inserito dovrà essere memorizzato in un *file* di nome `testo.txt`, che dovrà essere sovrascritto se già esistente.

Si tratta evidentemente di eseguire operazioni di **scrittura** su un *file*, quindi si userà la funzione `write`.

Inoltre, poiché il *file* dovrà essere sovrascritto se già esistente, si userà la modalità `"w"`.

Lettura/scrittura su *file*: esercizi

Una possibile soluzione (*file*: `72_scrivi_testo.py`)

```
print("Inserire una o più righe di testo,")
print("   e una riga vuota per concludere:")
f = open("testo.txt", "w")
riga = input()
while riga != "":
    f.write(riga + "\n")
    riga = input()
f.close()
```

Poiché `input` **non** include il carattere *newline* nella stringa restituita, tale carattere deve essere aggiunto nella scrittura sul *file*. In caso contrario l'intero testo inserito dall'utente verrebbe scritto nel *file* senza interruzioni di riga (si provi a sostituire la chiamata `f.write(riga + "\n")` con `f.write(riga)` e si osservi l'effetto).

Per verificare il corretto funzionamento del programma, dopo averlo eseguito si apra il *file* `testo.txt` con un *editor* come `Blocco note`.

Lettura/scrittura su *file*: esercizi

Una soluzione alternativa (*file*: `73_scrivi_testo_2.py`): si costruisce per concatenazione una **singola** stringa (a partire da una stringa vuota) contenente il testo inserito dall'utente, e la si scrive nel *file* con un'unica chiamata di `write`

```
print("Inserire una o più righe di testo,")
print("   e una riga vuota per concludere:")
testo = ""
riga = input()
while riga != "":
    testo = testo + riga + "\n"
    riga = input()
f = open("testo.txt", "w")
f.write(testo)
f.close()
```

Lettura/scrittura su *file*: esercizi

Si vuole definire una funzione che riceva come argomento una matrice di dimensione qualsiasi, rappresentata da liste nidificate (come si è visto in un esempio precedente), e la scriva in un nuovo *file* di nome `matrice.txt` (sovrascrivendolo se già esistente).

Per esempio, la matrice

$$\begin{pmatrix} 3 & -1 \\ 0 & 4 \end{pmatrix}$$

sarebbe rappresentata dalla lista `[[3, -1], [0, 4]]`.

Ogni riga della matrice dovrà essere scritta su una riga distinta del *file*, separando i suoi elementi con un carattere di spaziatura. Per esempio, il contenuto del *file* corrispondente alla matrice mostrata sopra sarebbe:

```
3 -1
0 4
```

Lettura/scrittura su *file*: esercizi

In questa soluzione (*file*: `74_scrivi_matrice.py`) si accede agli elementi della matrice mediante due istruzioni iterative nidificate; ciascuno di essi viene scritto nel *file* (seguito da un carattere di spaziatura) con una chiamata distinta di `write`.
Al termine di una riga della matrice (cioè al termine di ogni iterazione nidificata) si scrive nel *file* il carattere *newline*.
Si che ogni elemento della matrice è un valore di tipo numerico, e deve quindi essere convertito in una stringa, per mezzo della funzione `str`, prima di scriverlo nel *file* con `write`.

```
def matrice(m):
    f = open("matrice.txt", "w")
    for riga in m:
        for elemento in riga:
            f.write(str(elemento) + " ")
        f.write("\n")
    f.close()
```

Lettura/scrittura su *file*: esercizi

Si vuole scrivere un programma che stampi sulla *shell* tutte le parole contenute in un *file* di nome `testo.txt`, ciascuna in una riga **diversa** della *shell*. Per "parola" s'intende una qualsiasi sequenza di caratteri compresa tra caratteri di spaziatura o interruzioni di riga.

Per esempio, se il contenuto del *file* fosse:

```
Questo è
un esempio.
```

il programma dovrebbe stampare nella *shell*:

```
Questo
è
un
esempio.
```

Lettura/scrittura su *file*: esercizi

In questo caso non è necessario elaborare separatamente ciascuna riga del *file*: si userà quindi la funzione `read` per acquisirne il contenuto.

Per suddividere la stringa restituita da `read` nelle singole parole si userà la funzione `split`.

Ricordando che `split` restituisce una lista di stringhe corrispondenti alle singole parole, ciascuna parola dovrà essere stampata mediante un'iterazione su tale lista.

Lettura/scrittura su *file*: esercizi

Questa soluzione è disponibile nel *file* `75_stampa_parole.py`

```
f = open("testo.txt", "r")
testo = f.read()
f.close()
print("Il file contiene le seguenti parole:")
parole = testo.split()
k = 0
while k < len(parole):
    print(parole[k])
    k = k + 1
```

Lettura/scrittura su *file*: esercizi

Una versione alternativa dello stesso programma, nella quale si usa l'istruzione iterativa `for`

```
f = open("testo.txt", "r")
testo = f.read()
f.close()
print("Il file contiene le seguenti parole:")
parole = testo.split()
for parola in parole:
    print(parola)
```

Lettura/scrittura su *file*: esercizi

Come esempio della scelta tra le diverse funzioni di lettura da *file*, si supponga di voler riscrivere il programma precedente usando la funzione `readlines` invece che `read`.

Questo comporterebbe la scrittura di **due** istruzioni iterative **nidificate**: la prima operante sulla lista di stringhe (righe del *file*) restituita da `readlines`, l'altra (nidificata) per suddividere ogni stringa nelle singole parole e stampare queste ultime (si veda il programma mostrato di seguito).

La scelta di `readline` avrebbe conseguenze analoghe.

Si può concludere che in questo caso la funzione più appropriata è `read`, poiché essa consente di ottenere lo stesso risultato attraverso un programma più semplice.

Lettura/scrittura su *file*: esercizi

Lo stesso programma dell'esempio precedente, scritto usando la funzione `readlines`:

```
f = open("testo.txt", "r")
testo = f.readlines()
f.close()
print("Il file contiene le seguenti parole:")
for riga in testo:
    parole = riga.split()
    for parola in parole:
        print(parola)
```

Lettura/scrittura su *file*: esercizi

Ancora lo stesso programma dell'esempio precedente, scritto questa volta usando la funzione `readline`:

```
f = open ("testo.txt", "r")
print("Il file contiene le seguenti parole:")
riga = f.readline()
while riga != "":
    parole = riga.split()
    for parola in parole:
        print(parola)
    riga = f.readline()
f.close()
```

Lettura/scrittura su *file*: esercizi

Si consideri il *file* `matrice.txt` creato dal programma di uno degli esempi precedenti. Si vuole ora scrivere un programma che esegua l'operazione inversa, cioè acquisisca da tale *file* gli elementi della matrice e li memorizzi in una lista nidificata (ogni riga della matrice dovrà corrispondere a una lista nidificata).

In questo caso è conveniente usare la funzione `readlines`, dato che ciascuna riga del *file* dovrà essere elaborata separatamente (in particolare, dovrà essere memorizzata in una lista distinta).

Per ottenere da una data riga del *file* i valori (numerici) degli elementi della matrice si dovrà prima usare la funzione `split`, e poi convertire le stringhe corrispondenti in numeri frazionari usando la funzione `float`.

Saranno quindi necessarie due istruzioni iterative nidificate: quella principale accederà a ciascuna riga del *file*, quella nidificata elaborerà ciascuna riga e costruirà (per concatenazione) la lista corrispondente, inserendola poi (sempre per concatenazione) nella lista principale.

Lettura/scrittura su *file*: esercizi

Questa soluzione è disponbile nel *file* `76_leggi_matrice.py`:

```
f = open("matrice.txt", "r")
righe = f.readlines()
f.close()
m = []
for riga in righe:
    elementi = riga.split()
    riga_m = []
    for valore in elementi:
        riga_m = riga_m + [float(valore)]
    m = m + [riga_m]
print("La matrice è:\n", m)
```

Lettura/scrittura su *file*: esercizi

Si vuole definire una funzione che calcoli il numero di occorrenze di ciascuna delle cinque vocali (`a`, `e`, `i`, `o`, `u`) contenute in un *file* di testo, il cui nome sia l'argomento della funzione. Per semplicità si assume che il *file* contenga solo lettere minuscole.

Si deve definire anche un'opportuna struttura dati per memorizzare il valore che la funzione deve restituire.

Lettura/scrittura su *file*: esercizi

La funzione di lettura da *file* più appropriata è in questo caso `read`.

Una possibile struttura dati per memorizzare il risultato richiesto è un dizionario con cinque coppie chiave–valore corrispondenti alle cinque vocali. Le chiavi potranno essere `"a"`, `"b"`, ecc., mentre il valore associato a ciascuna di esse sarà il corrispondente numero di occorrenze.

Per esempio, se un *file* contenesse 3 occorrenze della lettera `a`, 5 della `e`, 2 della `i`, 3 della `o` e nessuna della `u`, il dizionario da restituire sarebbe

```
{"a": 3, "e": 5, "i": 2, "o": 3, "u": 0}
```

Lettura/scrittura su *file*: esercizi

Una soluzione elementare (*file*: `77_conta_vocali.py`)

```
def conta_vocali(nome_file):
    f = open(nome_file, "r")
    testo = f.read()
    f.close()
    vocali = {"a": 0, "e": 0, "i": 0, "o": 0, "u": 0}
    for lettera in testo:
        if lettera == "a":
            vocali["a"] = vocali["a"] + 1
        if lettera == "e":
            vocali["e"] = vocali["e"] + 1
        if lettera == "i":
            vocali["i"] = vocali["i"] + 1
        if lettera == "o":
            vocali["o"] = vocali["o"] + 1
        if lettera == "u":
            vocali["u"] = vocali["u"] + 1
    return vocali
```

Lettura/scrittura su *file*: esercizi

Per eseguire il programma (funzione) precedente si dovrà procedere come segue

- creare un *file* di testo, per esempio usando un *editor* come `Blocco note`, e inserire del testo al suo interno (scrivendo tutte le vocali come lettere minuscole)
- memorizzare il *file* contenente la definizione della funzione nella stessa *directory* del *file* di testo, ed eseguirlo
- chiamare la funzione dalla *shell*, indicando come argomento una stringa contenente il nome del *file* di testo; per es., se il nome di tale *file* fosse `testo.txt`, la chiamata sarebbe:

```
conta_vocali("testo.txt")
```

Lettura/scrittura su *file*: esercizi

Una soluzione più elegante si può ottenere tenendo conto che ciascuna delle stringhe che costituiscono le chiavi del dizionario coincide con la vocale corrispondente (*file*: `78_conta_vocali_2.py`)

```
def conta_vocali(nome_file):
    f = open(nome_file, "r")
    testo = f.read()
    f.close()
    vocali = {"a": 0, "e": 0, "i": 0, "o": 0, "u": 0}
    for lettera in testo:
        if lettera == "a" or lettera == "e" or \
           lettera == "i" or lettera == "o" or \
           lettera == "u":
            vocali[lettera] = vocali[lettera] + 1
    return vocali
```

Lettura/scrittura su *file*: esercizi

Nella soluzione precedente l'espressione condizionale dell'istruzione `if` si può scrivere in modo più conciso sfruttando la semantica dell'operatore `in` applicato alle stringhe

```
def conta_vocali(nome_file):
    f = open(nome_file, "r")
    testo = f.read()
    f.close()
    vocali = {"a": 0, "e": 0, "i": 0, "o": 0, "u": 0}
    for lettera in testo:
        if lettera in "aeiou":
            vocali[lettera] = vocali[lettera] + 1
    return vocali
```

Lettura/scrittura su *file*: esercizi

Si vuole definire una funzione che riceva come argomento il nome di un *file* che si assume contenere gli esiti di un esame; la funzione dovrà acquisire il contenuto del *file*, memorizzarlo in un'opportuna struttura dati, e restituire quest'ultima come risultato.

Si assume che ogni riga del *file* contenga le seguenti informazioni su **uno** studente, separate da un carattere di spaziatura: matricola, nome, cognome e voto. Il voto può essere 0 se la prova è insufficiente, oppure un numero tra 18 e 31, dove 31 significa "30 con lode". Per esempio, una riga del *file* può essere la seguente:

```
Maria Verdi 44444 27
```

Lettura/scrittura su *file*: esercizi

Una possibile struttura dati consiste in una lista di dizionari; ciascuno di essi corrisponderà a uno studente, e conterrà quattro chiavi associate a matricola, nome, cognome e voto. Si assume che la matricola debba essere codificata con una stringa (così come il nome e il cognome), e il voto con un numero intero. Il dizionario corrispondente all'esempio precedente sarà quindi:

```
{"nome": "Maria", "cognome": "Verdi",
 "matricola": "44444", "voto": 27}
```

La funzione di lettura più opportuna è `readlines`. Gli elementi di ogni riga del *file* (corrispondente a una stringa nella lista restituita da `readlines`) dovranno essere separati usando `split`; la stringa corrispondente al voto dovrà essere convertita in un numero intero mediante `int`.

La lista che la funzione dovrà restituire sarà costruita per concatenazione a patrire da una lista vuota; in modo analogo, ogni dizionario facente parte di tale lista sarà costruito inserendo all'interno di un dizionario inizialmente vuoto le quattro coppie chiave–valore.

Lettura/scrittura su *file*: esercizi

Una possibile soluzione (*file*: `79_esiti_esame.py`). Per verificarne il funzionamento si può usare il *file* allegato `esiti_esame.txt`.

```
def esiti_esame(nome_file):
    f = open(nome_file, "r")
    esiti = f.readlines()
    f.close()
    studenti = []
    for esito in esiti:
        riga = esito.split()
        studente = {}
        studente["matricola"] = riga[0]
        studente["nome"] = riga[1]
        studente["cognome"] = riga[2]
        studente["voto"] = int(riga[3])
        studenti = studenti + [studente]
    return studenti
```