

# Il linguaggio Python

Ing. Filippo Poltronieri  
[filippo.poltronieri@unife.it](mailto:filippo.poltronieri@unife.it)

Laboratorio di Reti

1

## Il Linguaggio Python













- Python è un linguaggio **orientato agli oggetti interpretato**, ideato da Guido Van Rossum nel 1991 ca.
- Python è un linguaggio utilzzatissimo, in svariati settori e lo *standard de-facto per applicazioni di machine learning*.
- Sono state rilasciate diverse versioni tra cui: 2, 2.7, 3, 3.6, ..,3.10
- Differenza fondamentale fra le macro versioni 2.+ e le versioni 3.+.
- Alcune versioni meno recenti come la 2.7 sono tuttora utilizzate, ma il loro supporto è terminato nel 2020.
- Nel corso, seguiremo le **regole definite dalla versione 3.+**, che rappresentano il riferimento più attuale.

2

## Il Linguaggio Python (popolarità secondo TIOBE) 2020

Dec 2020	Dec 2019	Change	Programming Language	Ratings	Change
1	2	▲	C	16.48%	+0.40%
2	1	▼	Java	12.53%	-4.72%
3	3		Python	12.21%	+1.90%
4	4		C++	6.91%	+0.71%
5	5		C#	4.20%	-0.60%
6	6		Visual Basic	3.92%	-0.83%
7	7		JavaScript	2.35%	+0.26%
8	8		PHP	2.12%	+0.07%
9	16	▲	R	1.60%	+0.60%
10	9	▼	SQL	1.53%	-0.31%
11	22	▲	Groovy	1.53%	+0.69%

## Il Linguaggio Python TIOBE Gennaio 2023

Feb 2022	Feb 2021	Change	Programming Language	Ratings	Change
1	3	▲	 Python	15.33%	+4.47%
2	1	▼	 C	14.08%	-2.26%
3	2	▼	 Java	12.13%	+0.84%
4	4		 C++	8.01%	+1.13%
5	5		 C#	5.37%	+0.93%
6	6		 Visual Basic	5.23%	+0.90%
7	7		 JavaScript	1.83%	-0.45%
8	8		 PHP	1.79%	+0.04%
9	10	▲	 Assembly language	1.60%	-0.06%
10	9	▼	 SQL	1.55%	-0.18%
11	13	▲	 Go	1.23%	-0.05%
12	15	▲	 Swift	1.18%	+0.04%

3

## Linguaggi Interpretati

I programmi scritti in un linguaggio di alto livello devono essere elaborati prima di poter essere eseguiti.

Questo processo di elaborazione impiega del tempo e rappresenta un piccolo svantaggio dei linguaggi di alto livello.

Essi vengono trasformati in programmi di basso livello eseguibili dal computer tramite due tipi di elaborazione:

- **L'interpretazione**
- **La compilazione**

4

# Linguaggi Interpretati

**Interpretazione:** è presente un interprete che legge il programma di alto livello e lo esegue, trasformando ogni riga di istruzioni in un'azione. L'interprete elabora il programma un po' alla volta, alternando la lettura delle istruzioni all'esecuzione dei comandi che le istruzioni descrivono:



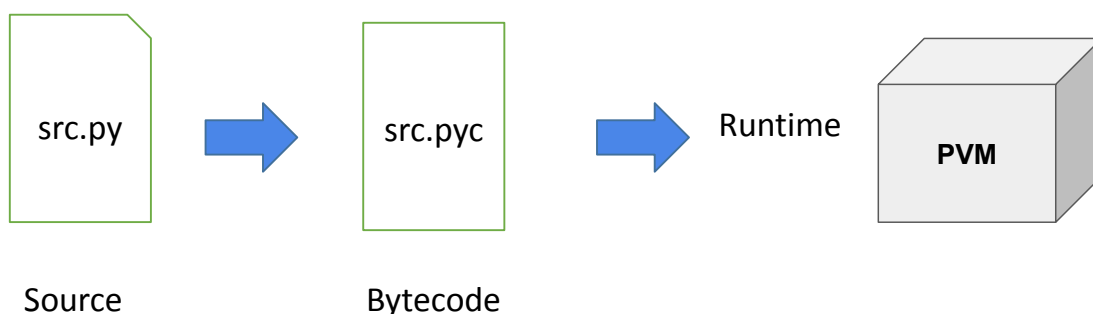
**Compilazione:** è presente un compilatore che legge il programma di alto livello e lo traduce completamente in basso livello, prima che il programma possa essere eseguito. In questo caso il programma di alto livello viene chiamato codice sorgente, ed il programma tradotto codice oggetto o eseguibile. Dopo che un programma è stato compilato può essere eseguito ripetutamente senza che si rendano necessarie ulteriori compilazioni finché non ne viene modificato il codice:



5

## Linguaggi Interpretati (il caso di Python)

Nel caso specifico di Python, ogni volta che viene invocato il comando `python`, il codice sorgente viene scansionato alla ricerca di token. Ogni token viene poi trasformato in una struttura logica ad albero che rappresenta il programma. Tale struttura viene, infine, trasformata in bytecode (file con estensione `.pyc` o `.pyo`). Per potere eseguire questi bytecode, si utilizza un apposito interprete noto come macchina virtuale Python (PVM).



6

## Editor di testo e IDE

Editor di testo: nel caso di un sistema operativo non Windows possiamo utilizzare un comune editor di testi per la scrittura di programmi in Python.

**Attenzione però a spaziatura e tabulazione!**

Infatti come vedremo, a differenza del C, Python identifica le aree di codice attraverso l'indentazione.

L'editor di testo utilizzato dovrà essere configurato per utilizzare 4 spazi per un tab (best practice): <https://google.github.io/styleguide/pyguide.html>

**Visual Studio Code** è un editor molto potente!

7

## Visual Studio Code

Visual Studio Code può essere scaricato direttamente dalla pagina principale del sito <https://code.visualstudio.com/>.

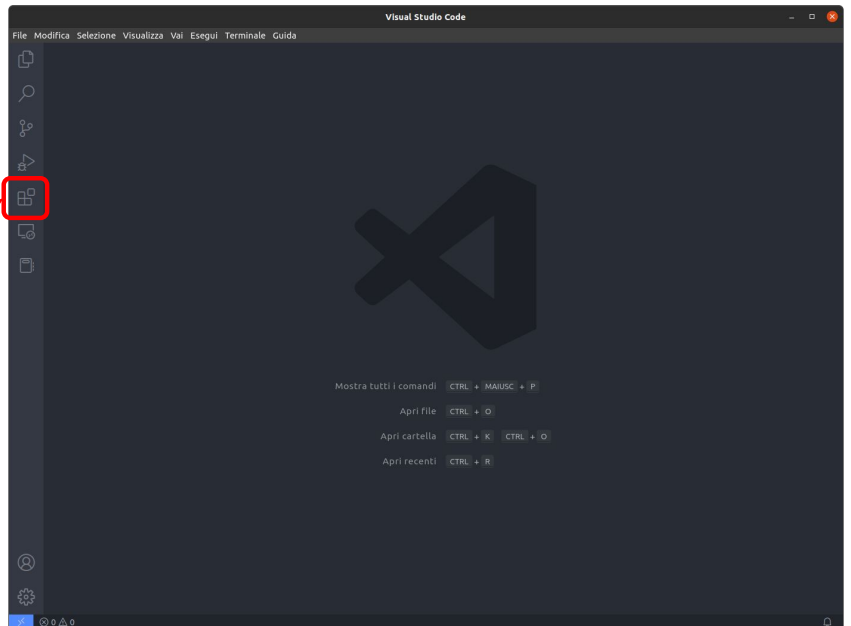
Code mette a disposizione tantissime estensioni, che ci permettono di migliorare la nostra produttività, come le funzionalità per il debugging e l'esecuzione del codice.

La gestione delle estensioni avviene direttamente tramite il menù di Code e la loro installazione non dovrebbe richiedere privilegi da amministratori.

8

# Installare le estensioni Python

Cliccare su estensioni

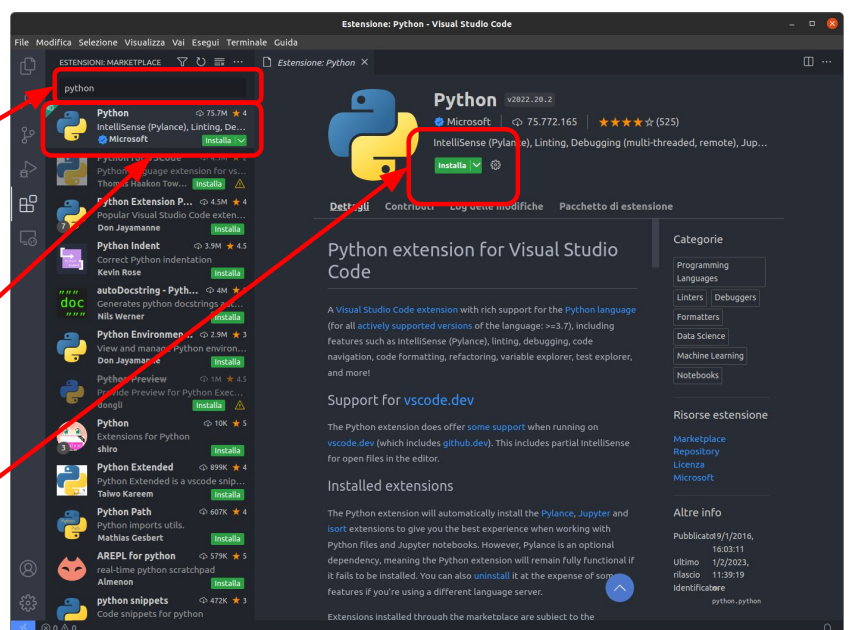


# Installare le estensioni Python

1) Cercare "Python"

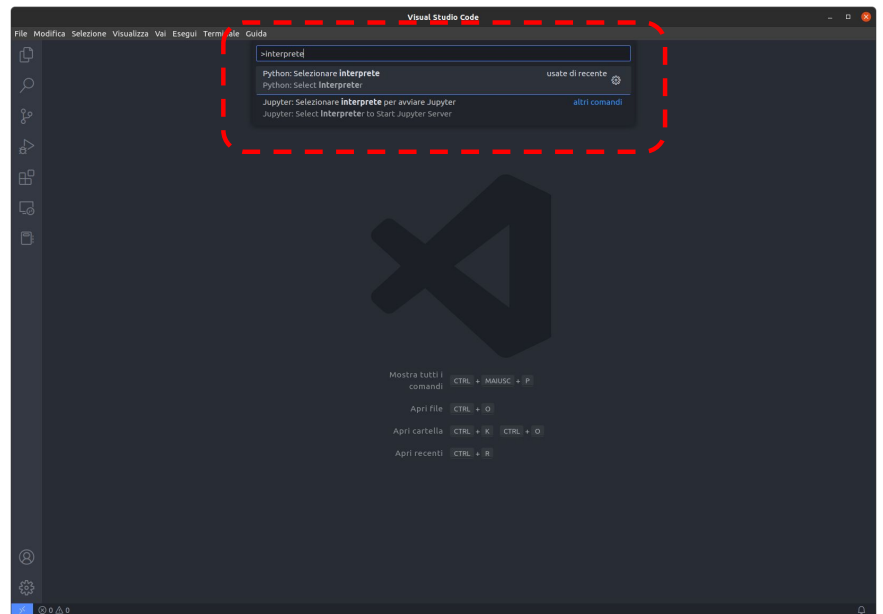
2) Cliccare sull'estensione "Python" di Microsoft

3) Cliccare "Installa"



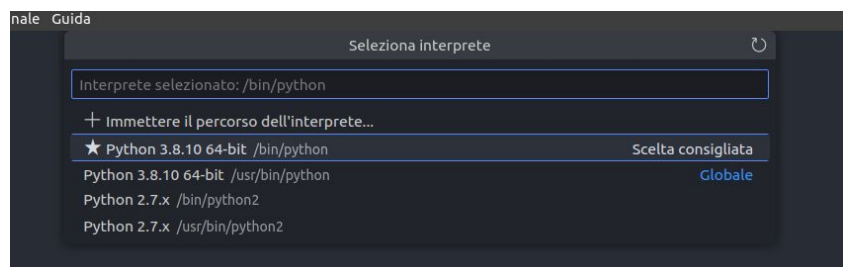
# Selezionare l'interprete Python

1. Con VSCode aperto, premere la combinazione di tasti CTRL+SHIFT+P
2. Scrivere "interprete" nella casella che compare
3. Selezionare (dando INVIO o cliccando con il mouse):  
"Python: Selezionare interprete"  
oppure, se il sistema è in Inglese, "Python: Select Interpreter"



## Selezionare l'interprete Python

- Comparirà una lista di interpreti disponibili nel sistema (in figura un esempio, sui PC del laboratorio sarà diversa)
- Selezionare:  
**Python 3.10.9 ('python3.10': conda)**

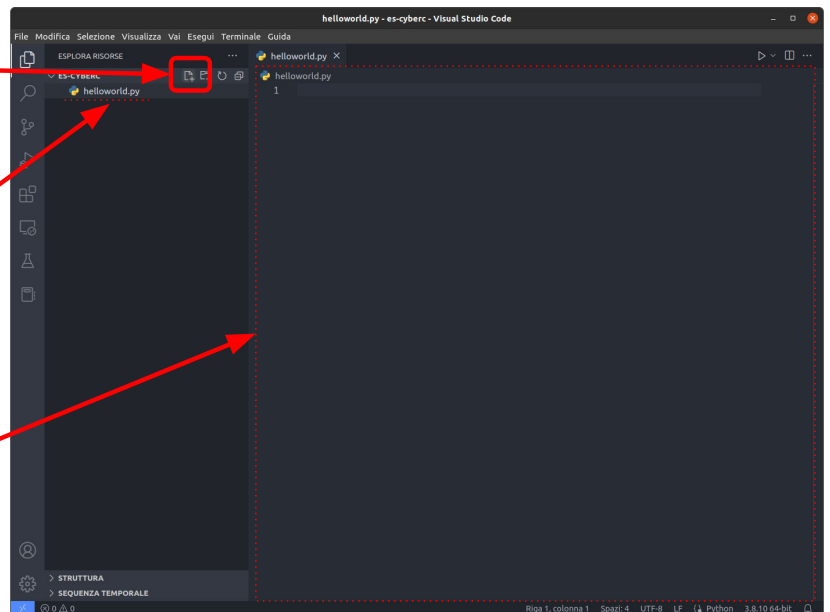


# Creare ed eseguire script Python con VSCode

1) Cliccare su  
"Nuovo file"

2) Dare un nome al file  
con estensione .py (es.  
helloworld.py)

Sulla destra potremo  
scrivere il codice dello  
script

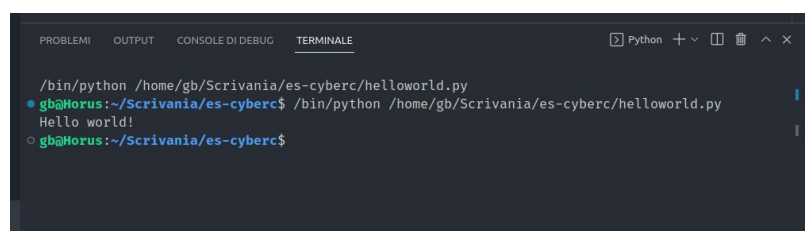
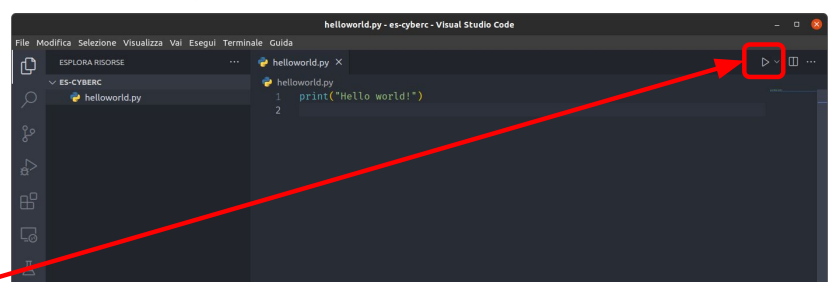


# Creare ed eseguire script Python con VSCode

1) Scrivere il codice, ad  
esempio un semplice:  
`print("Hello world!")`

2) Cliccare su "Esegui  
il file Python"

3) Si aprirà un terminale  
integrato a VSCode che  
eseguirà lo script Python



# Python: indentazione

## Python

```
if num > 5:
    print("out of range")
else:
    print("okay")
```

In python l'indentazione deve essere rispettata.

## C

```
if (num > 5) {
    printf("out of range\n")
}
else {
    printf("okay\n")
}
```

Diferentemente in C

## Utilizzo della shell Python interattiva

Per lanciare la shell di python nei sistemi Unix/Linux dobbiamo aprire un terminale e digitare il comando python.

- Nel caso in cui sul nostro sistema siano installate più versioni del linguaggio, ad esempio python2.7, python3.9, python3.10 dovremo lanciare il comando corrispondente.
- Una volta aperta la shell possiamo eseguire istruzioni scritte in linguaggio Python...

N.B. Ci sono strumenti utilissimi per gestire sia diverse versioni di Python sia le dipendenze di ogni progetto. Si guardino ad esempio [Conda](#) e pyenv.



# Esecuzione di un programma

- I programmi Python sono sequenze di istruzioni, e devono essere memorizzati in un file di testo. Per la scrittura di programmi si consiglia l'uso dell'editor di testi dell'ambiente di programmazione scelto (IDE), come [Visual Studio Code](#).
- Per convenzione, i file che contengono programmi Python hanno estensione .py
- Un programma può essere eseguito solo dopo averlo memorizzato in un file che può essere mandato in esecuzione direttamente da terminale o utilizzando l'IDE.

17

## Hello World in Python

```
print('Hello World!')
```

print stampa la stringa inserita  
sullo stdout;

a differenza del C non viene utilizzato il ";" a  
fine linea;

Non vi è bisogno di definire una funzione  
main per eseguire il print di una stringa o  
più istruzioni in sequenza;

18

# Hello World in Python (alternativa)

```
def main():  
    print('Hello World!')
```

Definizione di una  
funzione funzione  
main();

```
if __name__ == '__main__':  
    main()
```

all'avvio del programma eseguo la  
funzione main();

19

# Hello World in Python (alternativa)

```
def main():  
    print('Hello World!')
```

Definizione di una  
funzione funzione  
main();

```
if __name__ == '__main__':  
    main()
```

\_\_name\_\_ è una variabile speciale di Python che  
viene inizializzata in base a come eseguo lo script.  
Contiene il valore \_\_main\_\_ nel caso in cui mandi in  
esecuzione lo script. Conterrà il nome del file (in cui lo  
script è memorizzato) se importo lo script;

all'avvio del programma eseguo la  
funzione main();

20

# Inserire commenti nel codice

Ci sono due metodi per inserire dei commenti in un programma Python:

- `#` realizza un commento su una singola linea
- `"""` tre apici singolo per inserire un commento su più righe, va chiuso utilizzando altri tre singoli apici `"""`

In presenza di un commento, l'interprete python non esegue l'istruzione.

21

# Istruzioni

Generalmente si indica un'istruzione per linea. Inoltre, come abbiamo visto non dobbiamo utilizzare `;` al termine dell'istruzione.

Vi è la possibilità di scrivere più istruzioni per singola riga, separando queste istruzioni con `;`

Specificare più istruzioni in una singola righe è una pratica poco comune che peggiora la leggibilità del codice.

22

# Tipi di dato, variabili, espressioni

23

## I tipi di dato in Python

Nel linguaggio Python ogni cosa è un **oggetto**.

Un oggetto, è una istanza di una classe. Possiamo considerare un **oggetto** come un'astrazione di una entità dotata di uno *stato* e di un *comportamento*.

Lo *stato* di un oggetto di un determinato *tipo* memorizza il valore di tale oggetto.

Il *comportamento* di un oggetto specifica le operazioni che possiamo eseguire su/con quell'oggetto.

24

## I tipi di dato in Python

- Si pensi a un **oggetto di tipo “Rettangolo”**.
- Il suo **stato** è il valore della sua *base* e *altezza*.
- Il **comportamento** invece può essere l'insieme dei metodi che ci consentono di calcolare la sua base e la sua altezza.



25

## I tipi di dato in Python

- Invece, nel linguaggio C le cose sono differenti, si ragiona in termini di *variabili* (di un determinato tipo) e locazioni di memoria.
- In C, una variabile “punta” a una specifica locazione di area di memoria, che ha dimensione definita (in termini di bytes).
- La dimensione dipende strettamente dal tipo e alle volte dall'architettura del calcolatore
- Esempi sono variabili di tipo int (4 bytes), short (2 bytes), char (1 bytes), etc...
- Ciò comporta che in C il tipo della variabile debba essere specificato quando questa viene dichiarata. Questo non avviene in Python.

26

# I tipi di dato in Python

Python invece segue questi principi:

- un **oggetto** ha un **tipo specifico** (numero, stringa, dizionario etc..);
- invece una **variabile** è una etichetta (label) che fa riferimento a un oggetto;
- **una variabile in Python non ha tipo**;

27

## I tipi di dato in Python (gli oggetti in breve)

In Python gli oggetti sono delle **astrazioni per i dati** e vengono usati per rappresentare tali dati. Si può pensare ad un **oggetto** in python come un blocco di memoria che contiene il dato assegnatogli.

Un **oggetto** è un elemento attraverso cui è possibile operare in qualche modo sul dato che rappresenta. Un oggetto definisce:

❑ **Un'identità**: è un identificativo numerico che memorizza l'indirizzo di memoria dell'oggetto. Tale identità rimane unica e immutabile per tutta la durata del ciclo di vita dell'oggetto.

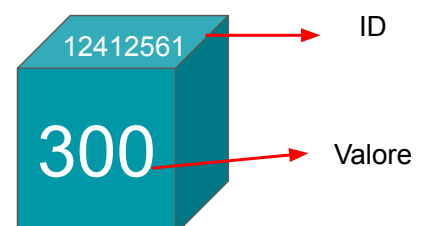
❑ **Un tipo**: indica che cosa rappresenta l'oggetto. Inoltre specifica:

❑ quali operazioni si possono eseguire sull'oggetto

❑ quali valori l'oggetto può assumere

❑ quali metodi l'oggetto mette a disposizione

❑ **Un valore**: dato manipolabile assegnato ad un oggetto



28

# Stringhe

Sequenze (*stringhe*) di caratteri, indicate tra apici singoli o doppi; es.: `"python"`, `'questa è una stringa'`. Le stringhe sono mappate in un oggetto di tipo **str**.

Possiamo inserire caratteri speciali in una stringa (apici singoli o doppi, *newline*, tabulazioni, ecc.), si usano sequenze di caratteri speciali, dette sequenze di *escape*, che iniziano con il carattere `\` (detto *backslash*).

Le principali sono:

- \`'` inserisce `'` in una stringa
- \`"` inserisce `"`
- \`\n` inserisce un *newline* ("andata a capo")
- \`\t` inserisce una tabulazione (sequenza di spazi)
- \`\\` inserisce `\` (lo stesso carattere *backslash*)      esempio: `"Prima  
riga\nSeconda riga"`

29

# Tipi di dato numerici

- **Numeri interi**; es.: `1, -5 --> 'int'`
- **Numeri reali**; es.: `-2.7, 3.14 --> 'float'`
- **Valori logici (*Booleani*)** "vero" e "falso", indicati con i simboli `True` e `False --> 'bool'`

30

# Dizionari - struttura ordinata

**Dizionario** (dict): insieme di elementi **ordinati** (da Python 3.7) costituito da coppie chiave - valore.

I dizionari sono ordinati nel senso che “ricordano” e **mantengono** l’ordine di inserimento degli elementi.

Bensì, due dizionari che contengono le stesse coppie chiave-valore ma in ordini differenti sono considerati **uguali**.

Se aggiungiamo un nuovo elemento a un dizionario, esso verrà aggiunto **alla fine** del dizionario, dopo tutti gli altri elementi.

I dizionari mantengono anche l’ordinamento degli elementi come specificato quando si crea un dizionario (ovvero, quando si fa un’operazione come quella sottostante).

```
mydiz = {1: "a", 2: "b", 3: "c"}
```

## Operatori e metodi principali dei dizionari

<b>in</b>	cerca un elemento fra le chiavi del dizionario	<pre>my_dict = {"list1": [1,2], "list2": [3,4]}  "list1" in my_dict → True "List1" in my_dict → False "list2" not in my_dict → False</pre>
<b>pop(x)</b>	restituisce il valore associato alla chiave x, poi rimuove l'elemento con chiave x (se non esiste la chiave x, dà errore)	<pre>persona = {"nome": "Pippo", "eta": 91} nome = persona.pop("nome") print(nome) → "Pippo" print(persona) → {"eta": 91}</pre>
<b>popitem()</b>	rimuove e restituisce l'ultimo item in forma di tupla	<pre>persona = {"nome": "Pippo", "eta": 91} my_item = persona.popitem() print(my_item) → ("eta", 91)</pre>
<b>values()</b>	restituisce tutti valori di un dizionario (restituisce una classe speciale dict_values)	<pre>persona = {"nome": "Pippo", "eta": 91} valori = persona.values() print(valori) → dict_values(['Pippo', 91])</pre>
<b>keys()</b>	restituisce tutte le chiavi di un dizionario (restituisce una classe speciale dict_keys)	<pre>persona = {"nome": "Pippo", "eta": 91} chiavi = persona.keys() print(chiavi) → dict_keys(['nome', 'eta'])</pre>



# Operatori e metodi principali dei dizionari

<b>items()</b>	restituisce tutti gli item di un dizionario (tutte le coppie chiave-valore, restituisce una classe speciale dict_items, ogni elemento è una tupla)	<pre>persona = {"nome": "Pippo", "eta": 91} coppie = persona.items() print(coppie) → dict_items([('nome', 'Pippo'), ('eta', 91)])</pre>
<b>update(diz)</b>	aggiunge a un dizionario gli item del dizionario diz (se diz contiene un item con chiave già esistente nel dizionario originale, il relativo valore viene sostituito con quello di diz)	<pre>d1 = {'a': 10, 'b': 20, 'c': 30} d2 = {'b': 200, 'd': 400} d1.update(d2)  print(d1) → {'a': 10, 'b': 200, 'c': 30, 'd': 400}</pre>
<b>len(diz)</b>	restituisce la lunghezza del dizionario, ovvero il numero di item (coppie chiave-valore)	<pre>d1 = {'a': 10, 'b': 20, 'c': 30} d2 = {'b': 200, 'd': 400}  print(len(d1)) → 3 print(len(d2)) → 2</pre>

## Dizionari di dizionari

I valori dei dizionari possono contenere qualsiasi tipo di oggetto Python, quindi anche **altri dizionari** (sintassi equivalente).

```
lezione = {
    "data": "28/03/2023",
    "corso": "Lab. di Prog.",
    "docente": {
        "nome": "Marco",
        "cognome": "Rossi"
    },
    "tutor": {
        "nome": "Claudio",
        "cognome": "Bianchi"
    }
}
```

# Dizionari di dizionari

Possiamo avere dizionari innestati **a piacere**.

```
person = {
    "name": {
        "first": "John",
        "last": "Doe"
    },
    "age": 35,
    "address": {
        "street": {
            "name": "Main St",
            "number": 123
        },
        "city": "Anytown",
        "state": "CA",
        "zip": "12345",
    },
    "contact": {
        "email": "johndoe@example.com",
        "phone": "555-555-1212"
    }
}
```

# Dizionari di dizionari

```
person = {
    "name": {
        "first": "John",
        "last": "Doe"
    },
    "age": 35,
    "address": {
        "street": {
            "name": "Main St",
            "number": 123
        },
        "city": "Anytown",
        "state": "CA",
        "zip": "12345",
    },
    "contact": {
        "email": "johndoe@example.com",
        "phone": "555-555-1212"
    }
}
```

Che risultati forniscono le seguenti espressioni?

person["age"]

person["name"]

person["name"]["first"]

person["address"]["street"]["name"]

person["contact"]["phone"][0:3]

person["contact"]["email"].split('@')

person["contact"]["email"].split('@')[1].split('.')[0]

# Dizionari di dizionari

```
person = {
    "name": {
        "first": "John",
        "last": "Doe"
    },
    "age": 35,
    "address": {
        "street": {
            "name": "Main St",
            "number": 123
        },
        "city": "Anytown",
        "state": "CA",
        "zip": "12345",
    },
    "contact": {
        "email": "johndoe@example.com",
        "phone": "555-555-1212"
    }
}
```

Che risultati forniscono le seguenti espressioni?

`person["age"]` → 35

`person["name"]` → {"first": "John", "last": "Doe"}

`person["name"]["first"]` → "John"

`person["address"]["street"]["name"]` → "Main St"

`person["contact"]["phone"][0:3]` → "555"

`person["contact"]["email"].split('@')` → ["johndoe", "example.com"]

`person["contact"]["email"].split('@')[1].split('.')[0]` → "example"

## Esercizio (1/2)

Dato il seguente dizionario `japanese_food` contenente il menù di un ristorante giapponese:

```
japanese_food = {
    "sushi": {
        "description": "Rice topped with various ingredients.",
        "origin": "Edo period (between 1603 and 1868)",
        "popular_types": ["nigiri", "maki", "temaki"]
    },
    "ramen": {
        "description": "A noodle soup dish.",
        "origin": "Early 20th century",
        "popular_types": ["shoyu", "shio", "miso"]
    },
    "tempura": {
        "description": "Deep-fried seafood or vegetables.",
        "origin": "16th century",
        "popular_types": ["ebi", "nasu", "kakiage"]
    }
}
```

Potete fare  
copia-incolla  
dalla slide

Lo si inserisca in uno script Python, poi...

## Esercizio (2/2)

Si crei un nuovo dizionario `okonmiyaki`, che è un cibo giapponese dalle caratteristiche scritte di seguito, **rispettando la struttura** degli altri cibi presenti nel dizionario `japanese_food`:

- descrizione → A savory pancake.
- origine → 16th century
- varianti tipiche → Hiroshima-style, Osaka-style

e lo si **aggiunga al menù**.

Si stampi poi a video la lista dei **nomi** di tutti i piatti presenti nel menù.

Suggerimento: `list(x)` trasforma, se è possibile, `x` in una lista.

## Liste

**Liste**: *sequenze ordinate* di zero, uno o più valori *qualsiasi*, indicati tra parentesi quadre e separati da virgole, ciascuno dei quali è identificato dalla propria posizione nella sequenza (*indice*: un intero a partire da 0); es.:

- `[]` (una lista vuota)
- `[1, 2, 3]`
- `['a', -5.3]`
- `[8, [5, 1], 3]` (una lista che ne contiene un'altra)

# Tuple

Le **tuple** sono liste immutabili:

```
tuple_name=(item0,item1,item2,...)
```

Utilizzare le parentesi tonde nel definire una tupla è una best practice!

Esempi di una tupla di stringhe sono:

```
partecipanti_corso=('Mario', 'Carlo', 'Guido')
```

Posso definire anche una tupla di un solo elemento, ma utilizzando , per evitare ambiguità

(8+3) vs. (8+3,)

41

## Differenze tra tuple liste

La differenza principale tra **tuple** e **liste** è che le prime sono immutabili, mentre le seconde possono essere modificate senza dover creare un nuovo oggetto.

Esempio:

```
tupla_a = ('a', 'b', 'c')
```

se provo a modificare il primo elemento della tupla ottengo un **errore** !

```
tupla_a[0] = 'd'
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

42

# Set

I set, sono sequenze di elementi immutabili e non indicizzabili.

```
{item0,item1,item2,...}
```

```
Set_c = {"red", "green", "yellow"}
```

Un set non può avere valori duplicati e non si può accedere ai suoi elementi utilizzando l'operatore [].

Come per gli altri tipi di dato, anche i set possono contenere elementi di tipo diverso.

43

# Bytes

Un oggetto di tipo bytes è una sequenza ordinata di interi (int) da 0 a 255. Gli oggetti di tipo bytes vengono utilizzati quando si legge o si scrive da/su una sorgente binaria come file, socket o interfacce di rete.

Ci sono diversi modi per inizializzare un oggetto di tipo bytes, partendo da stringhe, interi, etc...

Ad esempio:

`b'Hello World'` → converte la stringa 'Hello World' in un oggetto di tipo bytes

```
bytes([4,55,66])
```

44

# Bytes

Si può passare da bytes a stringhe utilizzando il metodo `bytes.decode()`. Per il passaggio speculare si può invece utilizzare `str.encode()`.

Entrambi i metodi consentono di specificare un encoding, ad esempio “utf-8”.

Esistono inoltre oggetti di tipo **bytearray**, che sono invece mutable a differenza di bytes.

Un oggetto di tipo bytearray supporta metodi e operatori che modificano gli elementi dell’array.

```
ba = bytearray([33, 14, 21])  
ba[1] = 2 # supporta l’assegnamento
```

45

## Tipi di dato principali - None

Non propriamente un tipo, ma importante nell’ecosistema Python è l’utilizzo di **None**

**None** è una keyword utilizzata per definire un valore null o “no value at all”.

Possiamo dire che **None** è equivalente al NULL dei linguaggi C e Java e può essere utilizzato per assegnare un valore null a un oggetto, prima della sua inizializzazione vera e propria.

Possiamo trovare None anche come valore di ritorno di funzioni o metodi in caso di errori... **None checking!**

Ad esempio: `Speed = None`

46

# Tipi di dato - funzione type

Per ottenere il tipo di dato di una variabile Python ci mette a disposizione una funzione apposita **type (espressione)**.

```
a = "Ciao, mondo!"
```

```
type(a)
```

```
<class 'str'>
```

`str` è un tipo di dato stringa, ma è possibile eseguire la stessa operazione con qualunque altra espressione.

Se `num = 5`, `type(num)` ritornerà?

47

## Tipi di dato principali (recap)

Tipi di dato	Nome	Descrizione	Esempio
Intero	int	Intero di dimensione arbitraria	-42, 0, 1200, 9999999999999999999
Reale	float	Numero a virgola mobile	3.14, -0.5, 0.004
Stringhe	str	Usata per rappresentare testo	' ', 'stefano', "l'acqua"
Booleano	bool	Per valori veri o falsi	True, False
Liste	list	Una sequenza mutabile di oggetti	[ ], [1, 2, 3], ['Hello', 'World']
Dizionari	dict	Una struttura che associa chiavi a valori (mappe tra oggetti)	{ }, {'nome': 'Ezio', 'cognome': 'Melotti'}
Nulla/None	NoneType	Un tipo di dato che contiene un valore None	result = None

48



## Oggetti: mutabili e immutabili

Non tutti gli oggetti Python gestiscono le modifiche allo stesso modo. Alcuni oggetti sono **mutabili**, nel senso che possono essere modificati senza che venga creato un nuovo oggetto.

Altri sono **immutabili**; non possono essere modificati, ma piuttosto restituiscono nuovi oggetti quando si tenta di aggiornare il loro valore.

Class	Immutabile?
bool	yes
int	yes
float	yes
<b>list</b>	<b>no</b>
tuple	yes
str	yes
<b>dict</b>	<b>no</b>

49

## Variabili

Le *variabili* sono nomi simbolici scelti dal programmatore (per es.: `x`, `area`), a ciascuno dei quali è possibile associare un valore *qualsiasi*, attraverso l'istruzione di assegnamento:

```
a = 5
```

Nei linguaggi di alto livello le variabili svolgono la stessa funzione delle celle di memoria nel linguaggio macchina.

50

# Operatori ed espressioni

Per i diversi tipi di dato sono disponibili *operatori* (per es., operatori aritmetici) che consentono di scrivere *espressioni*, ciascuna delle quali coinvolge uno o più valori di tipi appropriati, e produce un valore come risultato della sua valutazione

I nomi delle variabili alle quali sia già stato assegnato un valore possono essere usati all'interno di espressioni

In particolare, gli operatori *logici* consentono di scrivere *espressioni condizionali*, che consistono nel *confronto* tra coppie di valori o espressioni *qualsiasi*, e producono un valore logico (*Booleano*)

51

## Operatori principali su stringhe, liste e tuple

È possibile accedere ai singoli elementi di una stringa, di una lista o di una tupla memorizzata in una variabile, attraverso la seguente espressione:

**var** [**indice**]

- . **var** indica il nome della variabile
- . se indichiamo con **L** la lunghezza della sequenza, **indice** dev'essere una qualsiasi espressione il cui valore sia un numero intero compreso tra [-L, L-1]. Risulta quindi possibile accedere all'ultimo elemento di una sequenza con var[-1].

52

## Operatori aritmetici principali

- + addizione
- sottrazione
- \* moltiplicazione
- / divisione (se entrambi gli operandi sono interi, calcola la parte intera del quoziente)
- // divisione (calcola sempre la parte intera del quoziente)
- % modulo (resto della divisione)
- \*\* elevamento a potenza

53

## Operatori principali su stringhe e liste

Per esempio, assumendo che le variabili `s`, `L` e `i` contengano rispettivamente la stringa `'Python'`, la lista `['a', 'b', 'c']` e il numero 1:

- `s[0]`           ⇒ `'P'`
- `s[i]`            ⇒ `'y'`
- `L[i+1]`         ⇒ `'y'`
- `L[3]`           ⇒ produce un errore (il quarto elemento della lista `L`, cioè l'elemento di indice pari a 3, non esiste)

54

## Operatori principali su stringhe, liste e tuple

+ concatenazione

Esempi:

- 'a' + 'b'	⇒	'ab'
- 'abc' + 'de'	⇒	'abcde'
- [1,2] + [3]	⇒	[1,2,3]
- [1,2,3] + [4,5]	⇒	[1,2,3,4,5]
- ('a', 'b') + ('c', 'd')	⇒	('a','b','c','d')

55

## Altre operazioni sui tipi di dato

- Ogni tipo di dato mette a disposizione una serie di metodi che possono essere utilizzati su un oggetto di un determinato tipo.
- Questi **metodi sono strettamente dipendenti dal tipo di dato**.
- In una lista potremmo avere bisogno di inserire o rimuovere un elemento, o trovare il minimo (min) o il massimo (max).

56

## L'operatore del per le liste

- Per rimuovere un elemento da una lista utilizzando il suo indice si può utilizzare l'operatore **del**.
- **del** non restituisce un valore.
- **del** può essere utilizzato anche per cancellare un'intera lista.
- Esempi:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a restituisce []
```

57

## L'operatore del per le variabili

**del** può essere anche utilizzato per eliminare una variabile

```
>>> del a
```

Se cerchiamo di utilizzare la variabile `a` dopo averla eliminata si verifica un errore, ad esempio:

```
>> a = 'Hello'
>> del a
>> print(a)
```

Essendo `a` non più definita otteniamo il seguente errore:

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

58

## Operatori logici principali su valori numerici

Si applicano a coppie di espressioni, e producono un valore *Booleano* (`True` o `False`):

`==` uguaglianza

`!=` disuguaglianza

`<` minore

`<=` minore o uguale

`>` maggiore

`>=` maggiore o uguale

59

## Operatori principali su stringhe, liste e tuple

L'operatore di *slicing* si applica a variabili che contengono stringhe o liste, e restituisce una sottosequenza della sequenza originale

`var[a:b]` restituisce la sottosequenza (stringa o lista) contenuta nella variabile `var`, avente indici da `a` a `b-1`

`var[a:b:s]` restituisce la sottosequenza di indici `a`, `a+s`, `a+2s`, ..., fino all'elemento di indice `b` **escluso**

`var[:]` restituisce una *copia* dell'intera sequenza Es.: se `x` contiene

`'abcdefg'`:

`x[2:5]` restituisce `'cde'`

`x[0:7:2]` restituisce `'aceg'`

`x[:]` restituisce `'abcdefg'`

60

# Operatori logici principali su stringhe, liste e dizionari

`==` uguaglianza

`!=` disuguaglianza

`in` verifica la *presenza* di un elemento (una chiave nel caso dei dizionari), e restituisce `True` o `False`;

`not in` verifica l'*assenza* di un elemento; esempi:

```
'z' in 'python'=> False
```

```
1 in [1,2,3] => True
```

```
'b' in {'a':1, 'b':2} => True
```

```
'z' not in 'python' => True
```

```
1 not in [1,2,3] => False
```

```
'b' not in {'a':1, 'b':2} => False
```

61

## Metodi sulle liste

metodo	descrizione	esempio
<code>append(x)</code>	Aggiunge un elemento (x) alla fine della lista. L'equivalente si può ottenere nel seguente modo: <code>a[len(a):] = [x]</code> .	<pre>a = ["bee", "moth"] print(a) a.append("ant")</pre>
<code>extend(iterable)</code>	Estende la lista aggiungendo tutti gli elementi presenti in iterable. Con questo metodo è possibile unire due liste. L'operazione equivalente si ottiene nel seguente modo: <code>a[len(a):] = iterable</code> .	<pre>a = ["bee", "moth"] print(a) a.extend(["ant", "fly"])</pre>
<code>insert(i, x)</code>	Inserisce un elemento in una determinata posizione specificata da i.	<pre>a = ["bee", "moth"] a.insert(0, "ant") ["ant", "bee", "moth"] a.insert(2, "fly") ["ant", "bee", "fly", "moth"]</pre>

62

## Metodi sulle liste

metodo	descrizione	esempio
<code>remove(x)</code>	Rimuove il primo elemento con valore x dalla lista. Ritorna un errore se la lista non contiene tale elemento.	<pre>a = ["bee", "moth", "ant"] a.remove("moth")</pre>
<code>pop([i])</code>	Rimuove l'elemento alla posizione specificata e lo ritorna. Se i non viene specificato, <code>pop()</code> rimuove e ritorna l'ultimo elemento della lista.	<pre>a = ["bee", "moth", "ant"] print(a) a.pop() print(a)  a = ["bee", "moth", "ant"] print(a) a.pop(1) print(a)</pre>
<code>clear()</code>	Rimuove tutti gli elementi dalla lista. Equivalente a <code>del a[:]</code> .	<pre>a = ["bee", "ant", "moth", "ant"] a.clear()</pre>

63

## Metodi sulle liste

metodo	descrizione	esempio
<code>index(x[, start[, end]])</code>	Ritorna la posizione del primo elemento della lista che ha valore x. Se la lista non contiene x viene lanciato un <code>ValueError</code> . L'argomento opzionale end, viene utilizzato per limitare la ricerca in una porzione della lista.	<pre>a = ["bee", "ant", "moth", "ant"] print(a.index("ant")) print(a.index("ant", 2))</pre>
<code>count(x)</code>	Ritorna il numero di occorrenze di x all'interno della lista.	<pre>a = ["bee", "ant", "moth", "ant"] a.count("bee")</pre>
<code>sort(key=None, reverse=False)</code>	Ordina gli elementi della lista. Gli argomenti possono essere utilizzati per customizzare l'operazione di ordinamento. <b>key</b> Specifica una funzione di un argomento che viene utilizzata per estrarre una chiave di comparazione. Il default è None (gli elementi vengono confrontati direttamente). <b>reverse</b> Boolean. Se impostato a True, ogni comparison viene fatta al contrario.	Alcuni esempi nella slide a seguire

64



## Sort sulle liste - Esempi

```
a = [3,6,5,2,4,1]
a.sort()
print(a)
>> [1, 2, 3, 4, 5, 6]
-----
a = [3,6,5,2,4,1]
a.sort(reverse=True)
print(a)
>> [6, 5, 4, 3, 2, 1]
```

65

## Sort sulle liste - Esempi

```
a = ["bee", "wasp", "moth", "ant"]
a.sort()
print(a)
>> ['ant', 'bee', 'moth', 'wasp']

a = ["bee", "wasp", "butterfly"]
a.sort(key=len)
print(a)
>> ['bee', 'wasp', 'butterfly']

a = ["bee", "wasp", "butterfly"]
a.sort(key=len, reverse=True)
print(a)
>> ['butterfly', 'wasp', 'bee']
```

66

## E per tutti gli altri tipi di oggetti?

Esistono due funzioni integrate in python:

- **dir(<oggetto>)** → restituisce una lista di metodi e attributi dell'oggetto.
- **help(<oggetto>)** → restituisce una breve spiegazione riguardo all'oggetto passato come argomento.

Possiamo aprire una shell in python e consultare direttamente la documentazione disponibile per l'oggetto desiderato.

67

## Operatori logici su valori *Booleani*

Oltre a espressioni logiche *semplici* (consistenti in un confronto tra una coppia di valori) è possibile scrivere espressioni logiche *composte*, combinando espressioni semplici (o altre espressioni composte), attraverso tre operatori logici corrispondenti alle congiunzioni *e* e *o* (*oppure*) e all'avverbio *non* del linguaggio naturale.

Anch'esse producono il valore logico `True` o `False`:

- `and` (congiunzione di due espressioni logiche)
- `or` (disgiunzione inclusiva di due espressioni)
- `not` (negazione di una singola espressione)

68

# Esercizio su Operatori logici booleani

Per ognuna delle seguenti espressioni, le si memorizzi in una variabile, e subito di seguito (**e prima di eseguire lo script**) si scriva in un commento qual è il risultato che ci si aspetta venga restituito dall'espressione. Si stampino poi a video, in ordine, i valori di tutte le variabili, e si confronti se corrispondono al risultato che ci si aspettava di ottenere.

- not (True and False)
- (not True) or (not (True or False))
- not (not True)
- not (True and (False or True))
- not (not (not False))
- True and (not (not((not False) and True)))
- False or (False or ((True and True) and (True and False)))

## Istruzioni

# Istruzione di assegnamento

**variabile** = **espressione**

Assegna a una variabile, il cui nome è scelto dal programmatore, il valore di un'espressione *qualsiasi*, che può contenere variabili tra gli operandi.

*I nomi delle variabili possono contenere un numero qualsiasi di caratteri alfabetici, di cifre, e di simboli \_ (underscore), ma non possono cominciare con una cifra, né coincidere con le parole chiave del linguaggio*

71

# Istruzione di assegnamento

Esempi:

```
. i = 1
. s = 'Python'
. L = [1,2,3]
. x = L[i]
. j = i
. i = i + 1
```

Nota: l'espressione a destra del simbolo = viene calcolata *prima* dell'assegnamento alla variabile, quindi la conseguenza dell'ultima istruzione è incrementare di una unità il valore associato alla variabile *i* (assumendo che tale valore sia un numero)

72

## Note sull'assegnamento

Se però una variabile contiene una lista o un dizionario, e il suo valore viene assegnato a un'altra variabile, ogni modifica ai *singoli elementi* della lista o del dizionario attraverso una successiva istruzione di assegnamento a una delle due variabili *modificherà anche il valore dell'altra variabile*.

Per esempio, dopo le istruzioni:

```
a = [1, 2, 3]
b = a
a[0] = 10
```

sia il valore di `a` che quello di `b` saranno `[10, 2, 3]`

73

## Note sull'assegnamento

L'operatore di **slicing** applicato a una lista restituisce una *copia* della stessa lista. Può quindi essere usato per assegnare a una variabile una *copia* di una lista già assegnata a un'altra variabile. In questo caso, qualsiasi modifica degli elementi della lista attraverso un'istruzione di assegnamento a una delle due variabili **non modificherà l'altra variabile**.

Esempio:

```
x = [1, 2, 3, 4]
y = x[:]  y[0]=10
```

Ora il valore di `y` è `[10, 2, 3, 4]`, mentre quello di `x` è ancora `[1, 2, 3, 4]`

74

# Note sull'assegnamento

E' possibile assegnare a più variabili uno stesso riferimento, ad es.

```
a = b = c = 0
```

a, b e c fanno riferimento allo stesso oggetto, possiamo controllare l'id.

Esiste anche il caso particolare dell'unpacking

```
a, b, c = x
```

x deve essere un oggetto di tipo iterable (lista, ...) con esattamente 3 elementi, ad esempio:

```
x = [1, 2, 3]
```

```
a, b, c = x
```

75

# Note sull'assegnamento

Possiamo utilizzare l'unpacking anche per fare uno swap delle variabili, senza appoggiarci a una variabile temporanea:

```
a, b = b, a
```

Possiamo fare l'unpacking di una lista x nel seguente modo

```
first, *middle, last = x
```

che è equivalente alla seguente istruzione

```
first, middle, last = x[0], x[1:-1], x[-1]
```

76

# Istruzioni di ingresso/uscita

## Ingresso:

- `input(testo)`  
stampa sullo schermo il messaggio `testo`, poi resta in attesa che l'utente scriva una *qualsiasi* sequenza di caratteri attraverso la tastiera, e dopo la pressione del tasto "Invio" restituisce tale sequenza all'interno di una stringa
- `input()`  
come sopra, ma non stampa nessun messaggio

Nota: questa istruzione può essere usata per bloccare l'esecuzione di un programma fino alla pressione del tasto "Invio", per es.:

```
input("Premi Invio per continuare")
```

77

# Istruzioni di ingresso/uscita

## Uscita:

- `print(espressione)`
- stampa sullo schermo il *valore* di `espressione` (un'espressione *qualsiasi*)
- `print(espressione1, ..., espressioneN)`
- stampa sullo schermo i *valori* di ciascuna espressione, separati da un carattere di spaziatura.
- `print('Results a: {} b:{}'.format(espressione, espressione))`
- `print('Il risultato e: ', espressione)`

78

## Esercizi

- 1) Si scriva un programma Python per calcolare l'area e il perimetro di un rettangolo. Il programma deve chiedere in input all'utente il valore di base e altezza e stampare poi a video i valori di area e perimetro.
- 2) Si scriva un programma Python, anche partendo dall'esempio precedente, che vada a calcolare l'area di un triangolo rettangolo, dati la sua base e la sua altezza.
- 3) Si faccia lo stesso per un cerchio di raggio  $r$ , dove  $r$  deve essere letto in input dall'utente. Il pi-greco è definito all'interno del modulo `math` a `math.pi`

79

## Lettura dei parametri di input (riga di comando)

A volte, al lancio di un programma si richiede che vengano specificate delle opzioni. Queste opzioni sono argomenti di input al programma che possono essere letti all'interno del codice.

Esempio:

```
python my_program.py param
```

In questo esempio viene mandato in esecuzione un programma fittizio `my_program.py` passandogli come parametro una stringa `"param"` che rappresenta un parametro del programma.

80



## Lettura dei parametri di input (riga di comando)

Come possiamo leggere questi parametri in python?

Esiste una libreria del linguaggio chiamata `sys`.

Per importare una libreria all'inizio del nostro script (o in una shell Python) devo includere la seguente istruzione:

```
import <nome_libreria>
//nel nostro caso
import sys
```

81

## Lettura dei parametri di input (riga di comando)

Come nel linguaggio C, i parametri di input sono resi disponibili in una lista chiamata `sys.argv[0]` rappresenta il nome del programma, quindi il primo parametro di input (se inserito) sarà `sys.argv[1]`.

Possiamo ricavare quanti parametri di input sono stati passati controllando la dimensione della lista. Ovviamente il numero dei parametri inseriti corrisponderà alla lunghezza della lista meno uno.

Tali parametri possono essere assegnati a variabili nel codice del nostro script. Vediamo in breve l'esempio di `my_program`.

82

## Lettura dei parametri di input (riga di comando)

```
import sys

# ricavo il numero di parametri inseriti
num_par = len(sys.argv) - 1
print("Sono stati inseriti ", num_par, " parametri")

# ricavo il parametro username
param_1 = sys.argv[1]
print("Parametro_1: ", param_1)
```

83

## If/elif/else

84

## if

```
if condizione:  
    istr_1  
    ...  
    istr_2
```

Se l'espressione logica (*condizionale*) **condizione** è vera, viene eseguita la sequenza **istr\_1** , ..., **istr\_2**

Le istruzioni della sequenza devono essere scritte con una indentazione rispetto alla parola chiave `if`.

Per le sequenze composte da una sola istruzione, si può utilizzare la forma:

```
if condizione: istruzione
```

85

## if

```
if condizione:  
    istr_1  
    ...  
    istr_n  
else:  
    istr_1  
    ...  
    istr_n
```

Se una delle sequenze (o entrambe) fosse composta da una sola istruzione, si può utilizzare la forma breve.

86

## if-else

```
if condizione:  
    istr_1  
    ...  
    istr_n  
else:  
    istr_1  
    ...  
    istr_n
```

Le parole chiave `if` e `else` devono essere indentate a sinistra

Le istruzioni delle due sequenze devono essere scritte con un'indentazione di uno o più caratteri rispetto a `if` e `else`

Se **condizione** è vera, viene eseguita la sequenza di istruzioni nel ramo *if*, altrimenti viene eseguita la sequenza definita nel ramo *else*.

87

## if-elif-else

```
if condizione:  
    istr_1  
    ...  
    istr_n  
elif condizione2:  
    istr_1  
    ...  
    istr_n  
else:  
    istr_1  
    ...  
    istr_n
```

Si possono specificare più condizioni con il costrutto **elif** (else if). In questo caso viene valutata la condizione **if**, se questa è falsa viene valutata la condizione di **elif** e nel caso in cui questa sia falsa vengono eseguite le istruzioni specificate nel blocco **else**.

88

# match-case statement

Dalla versione 3.10 di Python è stato introdotto il costrutto match-case, che corrisponde allo switch-case di altri linguaggi di programmazione. Nell'utilizzare questo costrutto si deve prestare particolare attenzione alla versione dell'interprete utilizzato, che deve essere  $\geq 3.10$ .

**match** prende in ingresso un'espressione e ne compara il valore con i pattern specificati in conseguenti case block

```
match espressione:
    case 1:
        Istruzioni
    case 2:
        Istruzioni
    case _:
        print("Default case")
```

89

# match-case statement

Utilizzando match viene eseguito solamente il primo pattern che fa match.

La wildcard `_` serve per definire un pattern di default, che verrà eseguito nel caso in cui tutti i pattern precedenti non siano risultati validi.

Un classico esempio in cui match trova applicazione è il controllo dello stato di una risposta HTTP.

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

90

# match-case statement

Possiamo utilizzare diversi literal in un singolo case

```
match status:
    case 401 | 402 | 403:
        return "not allowed"
    ...
```

E anche per altri casi più complessi:

<https://docs.python.org/3/tutorial/controlflow.html#match-statements>

91

## Esercizio

Scrivere uno script Python che prenda in input la password di un utente, e successivamente crei e stampi a video un booleano che indica se la password è sicura.

Una password è considerata sicura se:

- Contiene almeno 8 caratteri
- Non contiene la parola "password" **indipendentemente da maiuscole e minuscole**
- Contiene sia lettere che numeri

## Soluzione esercizio

```
# Chiedo all'utente di inserire una password
password = input("Inserisci una password: ")
# implementare i controlli della password inserita
lpass = len(password)

if lpass < 8:
    check_1 = False
else:
    check_1 = True

# upper() mette in maiuscolo tutte le lettere
tmp_pass = password.upper()
# in alternativa casefold()

print("Password in versione maiuscola:", tmp_pass)

# qui sto utilizzando upper()
if "PASSWORD" in tmp_pass:
    check_2 = False
else:
    check_2 = True
```

```
# contiene sia lettere che numeri
if password.isalnum() and not password.isalpha() and
not password.isnumeric():
    check_3 = True
else:
    check_3 = False
valida = check_1 and check_2 and check_3
print("La password è valida? ", valida)

# Oppure in alternativa
# Versione inline
password_valida = len(password) > 8 and ("password" not
in password.casefold()) and (password.isalnum() and not
password.isalpha() and not password.isnumeric())

print("Valida ?", password_valida)
```

## Esercizio Dizionari

Dato il seguente dizionario `veicoli`, che associa le targhe di veicoli ai proprietari:

```
veicoli = {
    'AA111AA': 'Giacomo',
    'BB222BB': 'Simon',
    'CC333CC': 'Luca',
    'DD444DD': 'Simone',
    'EE555EE': 'Alessandro'
}
```

Potete fare  
copia-incolla  
dalla slide

Se vi serve, potete  
creare delle variabili  
aggiuntive!  
(non è obbligatorio o  
richiesto)

Lo si inserisca in uno script Python e si crei un dizionario `veicolidue` che scambia le chiavi con i valori. Ad esempio, il primo elemento di `veicolidue` dovrà essere:

`'Giacomo': 'AA111AA'`.

Nota: Si consulti l'helper fornito dalla shell interattiva.

## Esercizio Dizionari 2

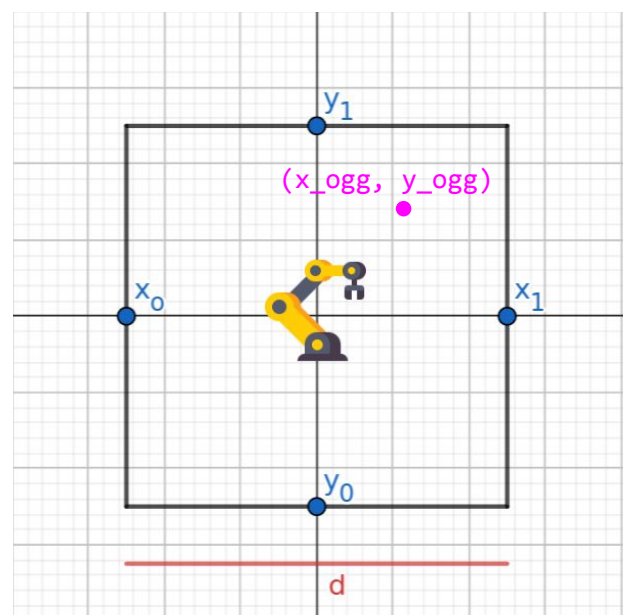
Creare uno script Python che usa un dizionario per gestire una "rubrica telefonica".

- Creare un dizionario rubrica che associa i nomi delle persone Mario e Luca ai rispettivi numeri di telefono (123-4567 e 789-0123)
- Aggiungere alla rubrica il numero di Anna, che è 345-6789332
- Stampare tutti i nomi presenti in rubrica
- Verificare la presenza del numero di Luca in rubrica
- Rimuovere il numero di Luca dalla rubrica
- Verificare nuovamente se il numero di Luca è in rubrica

## Esercizio if/elif/else

Un'azienda sta progettando un braccio robotico che sposta degli oggetti. Il braccio può raggiungere oggetti contenuti in un quadrato di lato  $d$ ; il braccio è posizionato al centro del quadrato, ed è al centro degli assi cartesiani.

Scrivere un programma Python che prende in input da tastiera due coordinate ( $x_{ogg}$  e  $y_{ogg}$ ) di un oggetto e stampa a video la variabile booleana che indica se il braccio robotico può raggiungere o meno tale oggetto.





# Il ciclo while

```
while condizione:
```

```
    istr_1  
    ...  
    istr_n
```

Esegue ciclicamente le istruzioni definite nella sequenza: se l'espressione logica **condizione** è vera, si esegue la sequenza istr\_1, ..., istr\_n; l'esecuzione termina non appena **condizione** diventa falsa.

Per il rientro e l'allineamento delle istruzioni della sequenza valgono le stesse regole dell'istruzione `if`.

97

# Il ciclo while

## Esempio

```
# stampa elementi lista con  
ciclo while  
a = [2, 3, 4, 5]  
# inizializzazione indice  
i = 0  
while i < len(a) :  
    print(a[i])  
    i += 1
```

Condizione di terminazione

Elemento i-esimo di a

incremento

## Output

2  
3  
4  
5

98

# Il ciclo for

```
for elemento in sequenza:
```

```
    istr_1
```

```
    ...
```

```
    istr_n
```

**sequenza** è un'espressione il cui valore dev'essere una stringa, lista o tupla

**elemento** è un nome di variabile scelto dal programmatore

Le istruzioni `istr_1` , ..., `istr_n` vengono eseguite per un numero di volte pari alla lunghezza di **sequenza**; nell'iterazione *i*-esima **elemento** ha come valore l'*i*-esimo elemento di **sequenza**

99

# Il ciclo for

## Esempio

```
l = [2, 3, 4, 5]
#ciclo for su lista
for i in l:
    print(i)
```

elemento i-esimo nella lista

## Output

2  
3  
4  
5

A ogni iterazione viene stampato l'elemento *i*-esimo nella lista `l`

100

## Cicli: forma breve

Sia per `while` che per `for`, se la sequenza è composta da una sola istruzione si può utilizzare la forma breve:

```
while condizione: istruzione
```

```
for elemento in sequenza: istruzione
```

101

## La funzione range

Serve a generare un insieme iterabile di numeri interi.

`range(i, j)` → `i, i+1, i+2, ..., j-1`

Il numero finale è **escluso**!

Si può specificare, oltre a inizio e fine, lo step (come terzo parametro).

Esempi: `range(5)` → `0, 1, 2, 3, 4`

`range(5, 10)` → `5, 6, 7, 8, 9`

`range(0, 10, 2)` → `0, 2, 4, 6, 8`

Nota: la funzione `range`  
**non** restituisce una lista!

```
>>> type(range(10))  
<class 'range'>
```

## La funzione range: esempio

```
# calcola il quadrato dei numeri da 1 a 10
squares = []

for value in range(1, 11):
    square = value ** 2
    squares.append(square)

print(squares)
```

Output: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

## Altri modi più eleganti, ma più complessi

### List comprehensions

```
squares = [i**2 for i in range(1,11)]
```

### Programmazione funzionale in Python!

```
squares = map(lambda x: x**2, range(1,11))
```

## Esempi di List comprehension

```
# Quadrati di tutti i numeri da 1 a 10
```

```
squares = [i**2 for i in range(1,11)] → [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
# Estrae solo i numeri positivi dalla lista
```

```
lista = [25,64,75,-23,-43,-9,52,-61,58,-1,10,0,-35]
```

```
solo_positivi = [x for x in lista if x > 0] → [25, 64, 75, 52, 58, 10]
```

## Esempi di List comprehension

```
# Moltiplica tutti i numeri della prima lista per tutti i numeri della seconda  
lista, solo se il primo numero è diverso da 0 e il secondo è diverso da 5
```

```
lista_uno = [2, 5, 0, -5, 10]
```

```
lista_due = [2, 3, 5]
```

```
lm = [x*y for x in lista_uno for y in lista_due if x != 0 and y != 5]
```

```
→ [4, 6, 10, 15, -10, -15, 20, 30]
```

## Esempi di List comprehension

<code>squares = [i**2 for i in range(1,11)]</code>	→	<code>squares = [] for i in range(1,11):     squares.append(i**2)</code>
<code>solo_positivi = [x for x in lista if x &gt; 0]</code>	→	<code>solo_positivi = [] for x in lista:     if x &gt; 0:         solo_positivi.append(x)</code>
<code>lm = [x*y for x in lista_uno for y in lista_due if x != 0 and y != 5]</code>	→	<code>lm = [] for x in lista_uno:     for y in lista_due:         if x != 0 and y != 5:             lm.append(x*y)</code>

## Il ciclo for: caso dict

```
for elemento in sequenza:  
    istr_1  
    ...  
    istr_n
```

Il valore di **sequenza** può anche essere un oggetto di tipo `<dict>`. In questo caso, in ognuna delle iterazioni **elemento** avrà come valore una delle *chiavi* del dizionario utilizzando una sequenza *ordinata* (dalla versione 3.7 di Python)

## Le istruzioni `break` e `continue`

Si possono usare solo all'interno di un ciclo (`while` o `for`)

- . `break` causa l'immediata conclusione dell'esecuzione dell'istruzione iterativa
- . `continue` causa il passaggio immediato all'iterazione successiva

109

## Esercizio 1

Si scriva uno script Python che, data una lista di stringhe, ad esempio:

```
lista = ["Carbonara", "Amatriciana", "", "Gricia", "", "Aglio Olio e  
Peperoncino", "Tonno", "", "Pesto", "", ""]
```

crei una nuova lista contenente solo le stringhe non vuote; infine, la stampi a video.

## Esercizio 2

Si scriva uno script Python che, data una lista di numeri, ad esempio:

```
lista = [10, 99, 98, 85, 45, 100, 59, 65, 66, 76, 12, 35, 13, 100, 80, 95]
```

stampi a video un messaggio ogni volta che, scorrendo la lista, incontra il numero 100, e in aggiunta stampi l'indice al quale l'ha trovato.

Ad esempio, con questa lista, un esempio di output corretto è:

```
Trovato 100 all'indice 5
```

```
Trovato 100 all'indice 13
```

## Esercizio 3

Si scriva uno script Python che prenda in input da tastiera un numero intero maggiore di 1, successivamente crei una lista `divisori` contenente tutti i divisori di tale numero, verificando tutti i numeri a partire da 2 fino al numero stesso -1.

Sulla base di tale lista, il programma deve stampare a video se il numero inserito da tastiera è primo o meno; se non lo è, deve stampare a video tutti i suoi divisori.

Visto che non verifichiamo come divisori i numeri 1 e il numero stesso, il numero inserito da tastiera sarà primo se non ha nessun divisore.



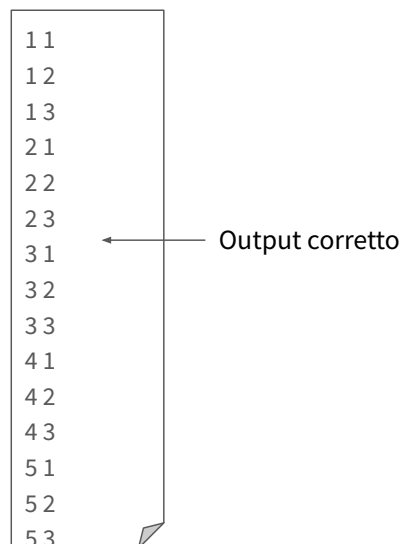
## Esercizio 4

Si scriva uno script Python che, dati due interi  $a$  e  $b$ , stampa tutte le coppie possibili di numeri  $x$  e  $y$  tali che:

$$\begin{cases} 1 \leq x \leq a \\ 1 \leq y \leq b \end{cases}$$

Per esempio, per  $a=5$  e  $b=3$  deve stampare l'output riportato a destra.

Si risolva questo esercizio usando due cicli for, uno innestato all'altro.



```
1 1
1 2
1 3
2 1
2 2
2 3
3 1 ← Output corretto
3 2
3 3
4 1
4 2
4 3
5 1
5 2
5 3
```

## Esercizio 5

Si scriva uno script Python che prenda in input da tastiera una frase, ad esempio "Oggi è il diciotto di Aprile 2023".

A partire da tale frase, crei una lista di parole `lista_parole`, dove ogni elemento della lista è una parola della frase.

Successivamente, mediante due cicli for (un ciclo for innestato all'altro), scorra parola per parola e lettera per lettera, e stampi a video quanto segue:

- Se la lettera corrente è compresa fra 'a' ed 'm' (entrambi inclusi) stampi la lettera così com'è
- Se la lettera corrente è compresa fra 'm' (non inclusa) e 'z' (inclusa) la stampi invece in maiuscolo
- In ogni altro caso stampi un trattino ( - )
- Alla fine di ogni parola stampi uno slash ( / )

Nella slide successiva si mostra un esempio di funzionamento.

## Esercizio 5 - Esempio di funzionamento

Inserisci una frase in minuscolo e senza lettere accentate: Oggi è il diciotto di Aprile 2023

```
-      d
g      i
g      /
i      -
/      P
-      R
/      i
i      l
l      e
/      /
d      -
i      -
c      -
i      -
O      /
T
T
O
/
```

Output corretto  
(è un unico  
output, scritto su  
due colonne solo  
per leggibilità)

Input dell'utente

## Esercizio 6

Si scriva uno script Python che crei, usando un ciclo while, una lista chiamata `lista` contenente tutti i numeri da 0 a 100.

A partire da tale lista, sempre usando un ciclo while, ne crei una seconda chiamata `solo_pari` che contiene solo i numeri pari contenuti in `lista`, e la stampi a video.

## Esercizio 7 (1/2)

Si scriva uno script Python che, usando un ciclo while, prenda in input da tastiera dei numeri interi (uno alla volta) e li aggiunga a una lista `numeri` finché l'utente non inserisce il numero 0.

Successivamente, usando un ciclo while, lo script deve calcolare e stampare a video la media **aritmetica** di tutti i numeri inseriti dall'utente.

$$\textit{Media aritmetica} = \frac{x_1 + x_2 + \dots + x_n}{n}$$

## Esercizio 7 (2/2)

Infine, sempre usando un ciclo while, lo script deve calcolare e stampare a video la media **geometrica** di tutti i numeri inseriti dall'utente.

$$\textit{Media geometrica} = \sqrt[n]{x_1 \cdot x_2 \cdot \dots \cdot x_n}$$

**Suggerimento:** fare la radice n-esima di un numero equivale a elevare tale numero al reciproco di n:

$$\sqrt[n]{a} = a^{\frac{1}{n}}$$

## Esercizio 8

Si scriva uno script Python che prenda in input da tastiera (una alla volta) delle parole, finché l'utente non inserisce FINE. Per ogni parola inserita, lo script dovrà avere il seguente comportamento:

- Se l'utente inserisce CLEAR, la lista di parole viene svuotata
- Se l'utente scrive COMPLETA, viene creata una frase a partire dalla lista di parole (ovvero, vengono concatenate le parole mettendo in mezzo a ognuna di loro uno spazio, ottenendo una stringa) e la frase viene stampata a video
  - Che metodo consente di fare questa operazione?
- Se la parola non è nè FINE, nè CLEAR, nè COMPLETA, la parola viene aggiunta alla lista di parole e tale lista viene stampata a video

Si veda un esempio di funzionamento nella slide successiva.

## Esercizio 8 - Esempio di funzionamento

```
Inserisci una parola: Oggi
Lista di parole inserite:
['Oggi']
```

```
Inserisci una parola: è
Lista di parole inserite:
['Oggi', 'è']
```

```
Inserisci una parola: il
Lista di parole inserite:
['Oggi', 'è', 'il']
```

```
Inserisci una parola: quattro
Lista di parole inserite:
['Oggi', 'è', 'il', 'quattro']
```

```
Inserisci una parola: Aprile
Lista di parole inserite:
['Oggi', 'è', 'il', 'quattro', 'Aprile']
```

```
Inserisci una parola: COMPLETA
La frase completa è: Oggi è il quattro Aprile
```

```
Inserisci una parola: CLEAR
Lista eliminata.
```

```
Inserisci una parola: ciao
Lista di parole inserite:
['ciao']
```

```
Inserisci una parola: a
Lista di parole inserite:
['ciao', 'a']
```

```
Inserisci una parola: tutti
Lista di parole inserite:
['ciao', 'a', 'tutti']
```

```
Inserisci una parola: COMPLETA
La frase completa è: ciao a tutti
```

```
Inserisci una parola: FINE
```

è un unico  
output, è scritto  
su due colonne  
per leggibilità

# Funzioni

121

# Funzioni

Le funzioni sono sequenze di istruzioni che svolgono una data operazione su un dato insieme di nessuno, uno o più valori (*argomenti*).

Ogni funzione è identificata da un **nome simbolico** (analogo ai nomi delle variabili).

L'esecuzione di una funzione su un certo insieme di argomenti può essere richiesta ogni volta che lo si desidera all'interno di un qualsiasi programma, attraverso un'opportuna espressione detta **chiamata**.

L'esecuzione di una funzione può produrre come risultato un valore, che può essere usato dal programma chiamante.

122

# Funzioni

Le funzioni vengono usate nei *linguaggi di alto livello* per realizzare operazioni di utilità generale all'interno di qualsiasi programma, **evitando ai programmatori di dover riscrivere le sequenze di istruzioni corrispondenti ogni volta che se ne presenti la necessità.**

Ogni linguaggio di programmazione comprende un insieme di funzioni *predefinite* o built-in che consentono di realizzare, per es., operazioni aritmetiche che gli operatori del linguaggio non rendono disponibili, come le funzioni matematiche del modulo *math* (log, sin, cos, ...)

123

## *Chiamata* di una funzione

L'esecuzione delle istruzioni di una funzione avviene attraverso la sua ***chiamata***, che può essere scritta come un'istruzione a sé stante, **oppure (se la funzione restituisce un valore)** come operando di un'espressione (inclusa un'espressione a destra del simbolo = in un'istruzione di assegnamento, per assegnare il valore di ritorno).

Sintassi di una chiamata a funzione:

```
>> function_do_stuff(argomenti)
>> val = function_do_stuff(argomenti)
```

**argomenti** è una sequenza di espressioni separate da virgole, il cui numero deve coincidere con quello degli argomenti della funzione

124

# Python standard library

Python definisce diverse funzioni predefinite (built-in), utilizzabili direttamente nel codice:

- `len(sequenza)` restituisce il numero di elementi di una stringa, lista o dizionario
- `range` con diverse modalità:
  - `range(a)` definisce la sequenza `0, 1, ..., a-1`, oppure una lista vuota se `a < 1`
  - `range(a, b)` restituisce la sequenza `a, a+1, ..., b-1`, oppure una sequenza vuota `a ≥ b`
  - `range(a, b, s)` restituisce una sequenza che contiene i valori `a, a+s, a+2*s, ...` fino a `b` escluso, oppure: una sequenza vuota se `a = b`, oppure `a < b` e `s < 0`, oppure `a > b` e `s > 0`; si verifica un errore se `s = 0`

125

# Python standard library

- `str(espressione)` restituisce una stringa che rappresenta il valore di `espressione`
- `int(stringa)` e `float(stringa)` restituiscono il numero intero e reale rappresentato da `stringa` (se questa è interpretabile come un numero)
- `int(numero)` restituisce la parte intera di un'espressione numerica
- `float(numero)` converte in un valore reale un'espressione numerica
- `abs(numero)` restituisce il valore assoluto di `numero`
- `min(lista)` e `max(lista)` restituiscono il minimo e il massimo valore contenuto in una lista
  - **Attenzione al loro utilizzo su liste con oggetti di diverso tipo!**

126

# Split di stringhe

`str.split()` restituisce una lista di stringhe ottenute suddividendo i caratteri di `str` in corrispondenza dei caratteri di spaziatura (che vengono rimossi); es.:

```
s = "Questa è una frase."  
s.split() ⇒ ['Questa', 'è', 'una', 'frase.']
```

`stringa.split(caratteri)`

dove `caratteri` è un'espressione avente per valore una stringa: come sopra, ma la suddivisione avviene in corrispondenza della sequenza `caratteri`; es.:

```
s = "pippo;pluto;paperino"  
s.split(';') ⇒ ['pippo', 'pluto', 'paperino']
```

127

# Utilizzo di librerie/moduli


Le funzioni di libreria/modulo richiedono un'istruzione particolare prima di poter essere usate (`import`). Per usare le funzioni definite in un file `nome.py`:

```
from nome import *
```

Per es., i file `math.py` e `random.py` contengono funzioni matematiche e funzioni per la generazione di numeri casuali. Per usare tali funzioni sono necessarie le istruzioni

```
from math import *  
y = sin(pi)
```

posso chiamare le funzioni senza utilizzare il nome della libreria come prefisso, e.g., `math.pi`



```
from random import *  
randint(10)
```

128



## Utility del modulo `math`

**cosine** `cos(x)`

**sine** `sin(x)`

**tangent** `tan(x)`

**degree/radian conversion** `radians(x)`, `degrees(x)`

math functions	<code>exp(x)</code>	math constants	
	<code>log(x)</code>		<code>pi</code>
	<code>log10(x)</code>		<code>e</code>
	<code>log(x,b)</code>		
	<code>pow(x,y)</code>		
	<code>sqrt(x)</code>		

129

## Modulo `random`

Il modulo `random` mette a disposizione funzioni per generare numeri casuali anche da diverse distribuzioni statistiche.

`random()`

genera un numero reale da una distribuzione uniforme nell'intervallo `[0,1]`

`uniform(a,b)`

genera un numero reale da una distribuzione uniforme nell'intervallo `[a,b]` (`a` e `b` sono numeri reali)

`randint(a,b)`

genera un numero intero casuale compreso tra `a` e `b` (`a` e `b` devono essere numeri interi)

`gauss(m,s)`

genera un numero casuale da una distribuzione Gaussiana con media `m` e deviazione standard `s`

130

## Modulo `random`

Le operazioni di `random` sulle liste:

`choice(sequenza)`

ritorna un elemento di `sequenza` (stringa o lista) scelto in modo casuale

`shuffle(lista)`

permuta in modo randomico gli elementi di una lista (*modificando* la lista originale)

`sample(sequenza, k)`

Ritorna una sottolista di dimensione `k` a partire scelti in modo casuale dalla `sequenza` (stringa, lista o dizionario). Nel caso di un dizionario, vengono restituite le *chiavi*

<https://python.readthedocs.io/en/stable/library/random.html>

131

## Esercizio 1

Si scriva un programma Python che generi in modo casuale una lista (o un set) di 10 numeri naturali casuali nell'intervallo  $[1, m]$ , dove `m` è un intero che deve essere letto in input da tastiera.

Una volta generata si stampi la lista a video

132

## Definire nuove funzioni

È possibile definire nuove funzioni, attraverso la seguente istruzione:

```
def nome_funzione (parametri) :  
    istr_1  
    ..  
    istr_n
```

**parametri** è una sequenza di nomi di variabili (separati da una virgola), alle quali saranno assegnati i valori degli *argomenti* della funzione, quando questa sarà *chiamata*.

Tutte le istruzioni della funzione devono essere indentate a destra.

133

## Definire nuove funzioni

I **parametri** possono avere anche un **valore di default**. Tale valore può non essere specificato durante la chiamata della funzione

```
def nome_funzione (param1, param2 = 'default'):  
    istr_1  
    ...  
    Istr_n
```

Per esempio, definiamo una funzione giocattolo che calcola l'area di un cerchio e che ha come parametro di default il raggio a 1..

```
def carea (radius = 1) :  
    area = math.pi * radius ** 2  
    return area
```

Vediamo in azione!

134

## L'istruzione `return`

Dopo l'esecuzione dell'ultima istruzione di una funzione, l'interprete riprende l'esecuzione delle istruzioni del programma chiamante

In un qualsiasi punto di una funzione è anche possibile usare l'istruzione **`return`** in due modi:

- . `return` termina l'esecuzione della funzione e restituisce il controllo al programma chiamante
- . `return` **espressione** termina come sopra, restituendo però il valore di **espressione** al programma chiamante (come nell'esempio `careia()`)

135

## L'istruzione `return`

```
def mia_funzione(n):  
    if n %2 == 0:  
        return n  
    else:  
        return n + 1
```

se invoco `mia_funzione`  
con `n=16` cosa ottengo?

e con `n=17`?

`return` può essere utilizzato in qualunque punto del codice della funzione.  
Una funzione può contenere più istruzioni `return`.

136

# Funzioni e variabili *locali*

Sia i parametri che le variabili eventualmente definite in una funzione sono **locali**, cioè sono "visibili" solo dalle istruzioni della stessa funzione, e non da altre funzioni o dal programma chiamante

A sua volta, una funzione non può accedere alle variabili usate dal programma chiamante

**È quindi sempre possibile usare nomi di variabili già in uso nel programma senza nessun conflitto o ambiguità!**

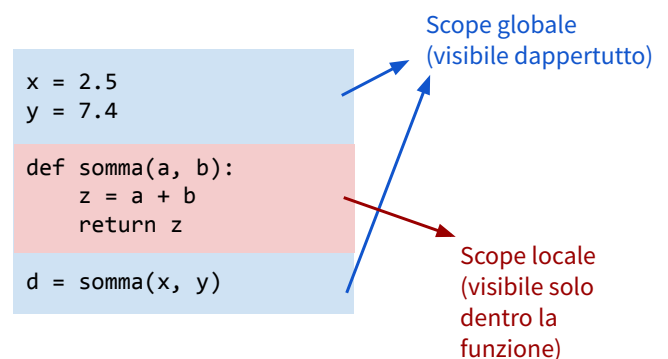
137

## Funzioni e variabili *locali*

Lo **scope** di una variabile definisce la zona del programma in cui la variabile è accessibile.

**Scope locale**, una variabile definita dentro una funzione ha visibilità solo all'interno di quella funzione.

**Scope globale**, una variabile non è definita dentro una funzione, quindi è visibile dovunque.



## Liste e dizionari come argomenti di funzioni

Per le liste e i dizionari le cose sono leggermente diverse. Per esempio:

```
def prova (lista):  
    lista[0] = 'a'
```

Se la funzione `prova` viene chiamata passandole come argomento una variabile contenente una lista:

```
x = [1,2,3]  
prova(x)
```

dopo la chiamata il valore di `x` sarà `['a', 2, 3]`.

139

## Liste e dizionari come argomenti di funzioni

Esempio:

```
def prova (lista):  
    lista[0] = 'a'
```

E se invece avessi passato a `prova` la copia di `x` (`x[:]`)?

```
x = [1,2,3]  
prova(x[:])
```

dopo la chiamata di `prova`, il valore di `x` sarà ? Vediamo...

140

## Liste e dizionari come argomenti di funzioni

Quando una funzione viene chiamata, i valori degli argomenti indicati nella chiamata **vengono assegnati ai suoi parametri, con le stesse modalità di un'istruzione di assegnamento**

Questo implica che se un argomento è una variabile che punta a una lista o un dizionario, la modifica dei suoi elementi da parte delle istruzioni della funzione si rifletterà anche sui valori della corrispondente variabile del programma chiamante.

Discussione interessante su StackOverflow:

<https://stackoverflow.com/questions/986006/how-do-i-pass-a-variable-by-reference>

141

## Uso di funzione in file esterni

Per usare le funzioni definite in un *file* `nome.py` (nella stessa cartella), si può utilizzare l'istruzione `import` seguita dal nome del file (senza estensione):

```
import nome_file
# uso funzione
nome_file.nome_funzione(param1, param2, ...)
Oppure:
from nome_file import *
# uso funzione
nome_funzione(param1, param2, ...)
```

142

## Funzioni Esercizio 1

Si scriva un programma Python che contiene la funzione per realizzare il calcolo del fattoriale di un numero  $n$  naturale, chiesto in input all'utente.

Si ricorda che il fattoriale di un numero  $n$  è dato dal prodotto di tutti i numeri da 1 a  $n$ ,  $n! = 1 \times 2 \times \dots \times (n - 1) \times n$ .

La funzione deve ritornare il risultato dell'operazione al programma chiamante.

Esempio: il fattoriale del numero 3,  $3! = 1 \times 2 \times 3 = 6$ .

143

## Funzioni Esercizio 2

Data una lista di numeri, ad esempio:

```
lista_num = [51, 92, 83, 65, 21, 47, 57, 1, 18, 70, 39, 11, 31, 59, 64]
```

si scriva uno script Python che definisca una funzione `crea_lista_doppi` che prenda in input la lista `lista_num`, e che restituisca una nuova lista dove ogni elemento è il doppio del corrispondente elemento di `lista_num`.

Successivamente, si stampi la lista dei numeri doppi a video.

Esempio di output corretto:

```
[102, 184, 166, 130, 42, 94, 114, 2, 36, 140, 78, 22, 62, 118, 128]
```



## Funzioni Esercizio 3

Si crei una funzione `calcola_stipendio` che prende come parametri il nome del dipendente, il suo stipendio base (numero) e la percentuale di bonus (intero da 0 a 100, **default 0**). La funzione deve calcolare e stampare a video lo stipendio totale del dipendente (non serve usare la `return`). Ad esempio, le seguenti chiamate alla funzione hanno come output:

<code>calcola_stipendio("Giacomo", 100000, 10)</code>	Lo stipendio totale di Giacomo è 110000.0 €
<code>calcola_stipendio("Simon", 100000)</code>	Lo stipendio totale di Simon è 100000.0 €
<code>calcola_stipendio("Luca", 150000)</code>	Lo stipendio totale di Luca è 150000.0 €
<code>calcola_stipendio("Axel", 80000, 50)</code>	Lo stipendio totale di Axel è 120000.0 €
<code>calcola_stipendio("Alessandro", 75000, 30)</code>	Lo stipendio totale di Alessandro è 97500.0 €

Notate che in due casi non è stato passato il terzo parametro, che è opzionale.

## Funzioni - Esercizio 4

Data una frase, ad esempio:

```
frase = "Oggi è il 3 di Maggio"
```

si scriva uno script Python che, definisca una funzione `trova_max_parola` che prenda in input la frase, e a partire da essa crei una lista di parole `lista_parole`. Poi, usando un ciclo `for` che scorra `lista_parole`, trovi la parola più lunga della frase e la restituisca (tramite una `return`). Infine, si stampi a video la parola più lunga della frase.

Esempio di output corretto:

Maggio

## Funzioni - Esercizio 5 (1/2)

Si vuole creare uno script Python che consenta di calcolare la somma dovuta a un dipendente per le sue ore di lavoro settimanale. Il programma deve prendere in input da tastiera, facendo uso di opportuni tipi di dato:

- Il nome del dipendente
- La sua paga oraria
- Il numero di ore che ha lavorato questa settimana

Si inseriscano i dati del dipendente in un appropriato tipo di dato che li contenga tutti.

Successivamente, si crei una funzione che calcola la paga totale settimanale del dipendente, tenendo a mente che le ore oltre le 40 sono considerate straordinari e che sono pagate il 50% in più della paga oraria. Continua...

## Funzioni - Esercizio 5 (2/2)

La funzione dovrà aggiungere alla struttura dati che è stata scelta per contenere i dati del dipendente la sua paga totale settimanale.

Infine, si stampino a video il nome e la paga totale settimanale del dipendente.

Esempio di output corretto:

Inserisci il nome del dipendente: **Giacomo**

Blu = input  
dell'utente

Verde =  
output

Inserisci la paga oraria del dipendente: **100**

Inserisci le ore che il dipendente ha lavorato questa settimana: **45**

La paga totale di questa settimana del dipendente Giacomo è **4750.0 €**

## Funzioni - Esercizio 6

Si crei un file Python contenente le funzioni necessarie a convertire miglia in chilometri, libbre in chili e gradi Fahrenheit in Celsius. Formule:

$$km = miglia \cdot 1.60934$$

$$kg = libbre \cdot 0.453592$$

$$C = (F - 32) \cdot \frac{5}{9}$$

Si supponga poi che vengano ricevute delle misurazioni da un'azienda americana che fa uso delle suddette unità di misura. Si ricevono più misurazioni per ognuna delle unità di misura. Una volta inserite le misurazioni in opportuni tipi di dato, si crei un nuovo file Python che faccia uso delle funzioni definite in precedenza per convertire e stampare a video tutte le misurazioni ricevute nelle unità di misura del Sistema Internazionale.

## Funzioni - Esercizio 7 (1/4)

Si vuole realizzare un programma Python per il controllo di qualità di un prodotto, che prenda in input una serie di misure effettuate su un componente e ne calcoli gli errori (in percentuale) rispetto al valore atteso. Si supponga che il prodotto debba misurare esattamente 10.0 centimetri per essere considerato perfetto, e che si sia deciso che l'errore massimo accettabile è del 5 %.

Un operatore misura il prodotto al termine del processo produttivo e inserisce le misure su un programma Python, e il programma mostra all'operatore alcune elaborazioni delle misure inserite.

(Continua nella prossima slide...)

## Funzioni - Esercizio 7 (2/4)

Si scelga il tipo di dato opportuno per le misure e per la struttura dati che le contiene

Si scriva un file Python contenente le seguenti funzioni (altre nella prossima slide):

- **crea\_misura()** → prende in input da tastiera, una alla volta, una misura effettuata dall'operatore e la aggiunge a una opportuna struttura dati. L'inserimento termina quando l'operatore inserisce 0. La funzione restituisce la struttura dati contenente tutte le misurazioni inserite dall'operatore.
- **calcola\_errore(misura, valore\_atteso)** → calcola e restituisce l'errore tra la misura e il valore atteso, che deve essere sempre un valore positivo.

Formula:

$$errore = \left| \frac{misura - val\_atteso}{val\_atteso} \cdot 100 \right|$$

## Funzioni - Esercizio 7 (3/4)

- **conta\_errori\_eccessivi(lista\_errori, max\_err\_accettabile)** → restituisce il numero di errori che superano la soglia massima accettabile.
- **calcola\_statistiche\_errori(lista\_errori)** → restituisce, in quest'ordine, la media, il massimo e il minimo degli errori. Si possono usare le funzioni `max()` e `min()` (e altre, se necessario).

Si scriva infine un altro file Python che fa uso delle funzioni che si sono definite per:

- creare la lista delle misure
- creare e stampare a video la lista di tutti gli errori
- calcolare e stampare a video il numero di errori non accettabili
- calcolare e stampare a video media, massimo e minimo degli errori

## Funzioni - Esercizio 7 (4/4)

Inserisci la misura letta o 0 per terminare: 10.2  
Inserisci la misura letta o 0 per terminare: 9.98  
Inserisci la misura letta o 0 per terminare: 9.995  
Inserisci la misura letta o 0 per terminare: 10.021  
Inserisci la misura letta o 0 per terminare: 12.84  
Inserisci la misura letta o 0 per terminare: 8.95  
Inserisci la misura letta o 0 per terminare: 10.04  
Inserisci la misura letta o 0 per terminare: 10.2  
Inserisci la misura letta o 0 per terminare: 9.64  
Inserisci la misura letta o 0 per terminare: 0

Blu = input  
dell'utente

Verde =  
output

Gli errori calcolati, in percentuale, sono:

[1.999999999999927, 0.1999999999999576, 0.050000000000007816, 0.21000000000000796, 28.4, 10.500000000000007, 0.3999999999999153, 1.999999999999927, 3.599999999999943]

Sono state trovate 2 misurazioni con errore maggiore del 5 %

La media degli errori è 5.262222222222215 %, l'errore massimo è 28.4 % e il minimo è 0.050000000000007816 %

## Gestione dei *file*

# Gestione dei *file*

Tipi di *file*:

- sequenze di caratteri (*file di testo*)
- sequenze di *byte* (*file binari*)

Operazioni eseguibili sui *file* di testo:

- lettura
- scrittura, con tre possibilità:
  - creazione di un nuovo *file*
  - aggiunta di dati al termine di un *file* esistente
  - sostituzione del contenuto di un *file* esistente (sovrascrittura)

155

# Gestione dei *file*

Procedura per l'accesso ai *file*:

- 1) apertura di un *file* in modalità di lettura o di scrittura
- 2) esecuzione di operazioni di lettura oppure di scrittura  
(a seconda della modalità scelta in fase di apertura)
- 3) chiusura del *file*

156

## Apertura di un *file*

`open (nome_file, modalità)` , restituisce un oggetto per operare sul *file*

**nome\_file**: stringa contenente il nome del *file* (*pathname*)

**modalità**:

- 'r' lettura da un *file* esistente
- 'w' scrittura: crea un nuovo *file* (se non esiste nessun *file* di nome **nome\_file**) o sovrascrive un *file* esistente
- 'a' scrittura: aggiunta di dati a un *file* esistente

NOTA: per i file di testo non vi è bisogno di apporre “t” alla modalità di apertura.

157

## Apertura di un *file*

Esempio (apertura del file dati.txt):

```
f = open ('dati.txt', 'r')
```

Con f posso svolgere tutte le operazioni (di lettura nel caso specifico).

63

158

## Chiusura di un *file*

Si usa la funzione predefinita `close`

```
f.close()
```

chiude il *file* associato a `f` (non sarà più possibile eseguire operazioni di lettura e scrittura su di esso, a meno di riaprirlo con la funzione `open()`)

159

## Lettura da un *file*

Python mette a disposizione diverse funzioni predefinite per acquisire da un *file* di testo sequenze di uno o più caratteri, una o più righe, o il suo intero contenuto. Un file viene letto come se fosse un nastro, in modo sequenziale.

Ogni operazione di lettura viene eseguita:

- **a partire dal primo carattere del *file***, se si tratta della prima operazione di lettura dopo l'apertura
- **a partire dal carattere successivo all'ultimo carattere letto**, se non è la prima operazione di lettura.

I dati acquisiti da ciascuna di tali funzioni vengono di norma assegnati a una variabile, o vengono usati all'interno di un'espressione

160



## Lettura da un *file*: `read`

La funzione `read` consente anche di acquisire un dato numero di caratteri da un file, invece che tutti i caratteri restanti, indicando tale numero come argomento:

```
file.read(n)
```

dove `n` è un'espressione il cui valore dev'essere un numero intero positivo. Per leggere l'intero contenuto del file va invece utilizzato:

```
file.read()
```

`read()` restituisce una stringa vuota nel caso in cui il file sia vuoto (o già letto interamente).

161

## Lettura da un *file*: `read`

La funzione `read` restituisce una stringa contenente:

- l'intero contenuto del *file*, se viene chiamata subito dopo l'apertura
- i caratteri non ancora acquisiti dalle operazioni di lettura precedenti (se tutti i caratteri sono già stati acquisiti, restituisce una stringa vuota)

Sintassi: `variabile_file.read()`

dove `variabile_file` è la variabile associata al *file*

162

## Lettura da un *file*: `readline`

La funzione `readline` restituisce una stringa contenente una singola riga di un *file*, cioè una sequenza di caratteri conclusa da un *newline* ("andata a capo", indicata con la sequenza di escape `'\n'`), a partire dal carattere successivo rispetto all'ultimo acquisito da eventuali operazioni di lettura precedenti. Note:

- se una riga è vuota, `readline` restituisce la stringa `'\n'`
- se tutti i caratteri sono già stati acquisiti, `readline`
- restituisce una stringa vuota

Sintassi:

```
variabile_file.readline()
```

O per leggere più linee completamente

```
lines = file_variable.readline(n)
```

163

## Lettura da un *file*: `readline`

Esempio vogliamo leggere un file intero, linea per linea.

Come fare?

```
file = open(filename, 'r')
line = file.readline()
while (line != ""):
    print(line)
    line = file.readline()
```

164

## Lettura da un *file*: `readlines`

La funzione `readlines` restituisce l'intero contenuto di un *file* (o della parte non ancora acquisita dopo le precedenti operazioni di lettura).

Ritorna una lista di stringhe, ognuna delle quali contiene una singola riga (incluso il carattere '`\n`')

```
list_of_lines = variabile_file.readlines()
```

165

## Scrittura su un *file*: `write`

La funzione `write` consente di scrivere in un *file di testo* una stringa di caratteri, indicata come argomento:

```
variabile_file.write(stringa)
```

- **stringa** deve essere un'espressione il cui valore sia un oggetto di tipo `<str>`
- la stringa viene scritta al termine del *file* (cioè dopo gli eventuali caratteri scritti da precedenti operazioni di scrittura)

166

# File Esercizio

Si implementi un programma python che deve creare una copia di backup di un file specificato in input dall'utente.

Il programma deve salvare la copia di backup in un nuovo file avente lo stesso nome del file di partenza e con suffisso ".bak".

167

## Oltre la singola operation mode

Modalità	Descrizione
'r'	Apri un file di testo in lettura. Modo di apertura di default dei file.
'w'	Apri un file di testo in scrittura. Se il file non esiste lo crea, altrimenti cancella il contenuto del file.
'a'	Apri un file di testo in <i>append</i> . Il contenuto viene scritto alla fine del file, senza modificare il contenuto esistente.
'x'	Apri un file di testo in creazione esclusiva. Se il file non esiste, restituisce un errore, altrimenti apre in scrittura cancellando il contenuto del file.
'r+'	Apri un file di testo in modifica. Permette di leggere e scrivere contemporaneamente.
'w+'	Apri un file di testo in modifica. Permette di leggere e scrivere contemporaneamente. Cancella il contenuto del file.

168

## Oltre la singola operation mode

Tra i diversi operation mode elencati nella tabella, `'r+'` **risulta essere molto interessante**. Posso aprire un file ed eseguire sia operazioni di lettura che operazioni di scrittura.

Un'altra metodo interessante su un oggetto di tipo file è il metodo `seek`.

Questo metodo ci permette di spostare **il cursore** a una posizione diversa nel file. Ricordiamoci che il cursore si sposta mano a mano che leggo o scrivo nel file.

Se apro un file in lettura il cursore si troverà alla posizione 0. Se poi leggo 10 caratteri dal file il cursore si troverà alla posizione 10.

169

## Spostarsi in un file

**Il metodo `seek`** va invocato su un oggetto di tipo file con la seguenti sintassi:

```
file.seek(offset, position)
```

`offset` indica di quanto ci dobbiamo spostare (anche 0) dalla posizione definita da `position`. Per `position` si utilizzano tre valori standard:

- 0 - per indicare l'inizio del file;
- 1 - per indicare la posizione corrente; solo per file aperti in modalità binaria;
- 2 - per indicare la fine del file;

**Il metodo `tell`:**

`file.tell()` restituisce la posizione corrente memorizzata dal file object.

170

# Spostarsi in un file

Alcuni esempi:

```
file.seek(0, 0)
```

mi sposta il cursore all'inizio del file;

```
file.seek(0, 2)
```

mi sposta il cursore alla fine del file;

```
file.seek(15, 1);
```

mi muove in avanti il cursore di 15 bytes dalla posizione corrente. E' possibile eseguire questa operazione solamente se il file è stato aperto in modalità binaria.

171

## Metodi per eliminare caratteri speciali

**Tabella 1**  
Metodi per eliminare  
caratteri da una stringa

Metodo	Restituisce
<code>s.lstrip()</code> <code>s.lstrip(caratteri)</code>	Una nuova versione di <i>s</i> in cui eventuali caratteri di spaziatura (spazi, caratteri di tabulazione e <i>newline</i> ) sono stati eliminati <i>a sinistra</i> , cioè all'inizio, di <i>s</i> (la lettera <i>l</i> di <i>lstrip</i> sta, appunto, per <i>left</i> , sinistra). Se è presente la stringa <i>caratteri</i> , vengono eliminati i caratteri presenti in essa invece dei caratteri di spaziatura.
<code>s.rstrip()</code> <code>s.rstrip(caratteri)</code>	Come <i>lstrip</i> , ma i caratteri vengono eliminati <i>a destra</i> , cioè alla fine, di <i>s</i> (la lettera <i>r</i> di <i>rstrip</i> sta per <i>right</i> , destra).
<code>s.strip()</code> <code>s.strip(caratteri)</code>	Simile a <i>lstrip</i> e <i>rstrip</i> , ma i caratteri vengono eliminati tanto a sinistra quanto a destra di <i>s</i> .

172

## Esempi

Enunciati	Risultato	Commento
<pre>string = "James\n" result = string.rstrip()</pre>	J a m e s	Il carattere <i>newline</i> alla fine della stringa è stato eliminato.
<pre>string = "James \n" result = string.rstrip()</pre>	J a m e s	Anche lo spazio alla fine della stringa è stato eliminato.
<pre>string = "James \n" result = string.rstrip("\n")</pre>	J a m e s	È stato eliminato soltanto il carattere <i>newline</i> .
<pre>name = " Mary " result = name.strip()</pre>	M a r y	I caratteri di spaziatura sono stati eliminati tanto all'inizio quanto alla fine della stringa.
<pre>name = " Mary " result = name.lstrip()</pre>	M a r y	I caratteri di spaziatura sono stati eliminati soltanto all'inizio della stringa.

173

## Oltre la singola operation mode

Di default i parametri vengono utilizzati per aprire **file testuali**, ma è possibile anche file in rappresentazione binaria.

Per aprire un file in forma binaria devo includere `b` all'operation mode desiderato. Per esempio:

`open(filename, 'rb')` apre filename in *lettura* e in modalità binaria

`open(filename, 'wb')` apre filename in *scrittura* e in modalità binaria

Invece, quando apriamo un file in modalità testuale non specifichiamo `t` in quanto già stabilito di default.

174

## Oltre la singola operation mode

### Che cosa cambia? (in lettura)

I metodi di lettura (come `read`) invocati su un file aperto in modalità binaria mi ritornano un oggetto di tipo `bytes` e non più `str`.

```
f = open(filename, 'rb')
rb = f.read(10) # leggo 10 bytes
type(rb)
>> <class 'bytes'>
```

175

## Oltre la singola operation mode

### Che cosa cambia? (in scrittura)

Se voglio scrivere su un file aperto in modalità binaria dovrò passare al metodo `write()` un oggetto di tipo `bytes` e non una stringa.

Il metodo più semplice per convertire una stringa in `bytes` è quello di utilizzare il metodo `bytes` con encoding UTF-8.

```
bytes('Ciao Mondo', 'utf-8')
```

Mi restituisce la rappresentazione in `bytes` della stringa 'Ciao Mondo' secondo l'encoding UTF-8.

176



## Il costrutto **with**

Questo costrutto può essere usato con dei context manager (manager di contesti), cioè degli oggetti particolari che specificano delle operazioni che vanno eseguite all'entrata e all'uscita del contesto.

Questa forma del **with** ci permette di creare l'oggetto direttamente e di assegnargli un nome dopo la keyword **as**. Tale variabile avrà visibilità locale solo all'interno del costrutto with.

Una volta usciti dal contesto il file verrà chiuso automaticamente!

```
file = open('<file>', '<condizione>')  
<operazioni sul file>  
file.close()
```



```
with open('<file>', '<condizione>') as file:  
    <operazioni sul file>
```

177

## Esempio lettura del file - read()

```
with open('dog_breeds.txt', 'r') as reader:  
    print(reader.read())
```

```
# Pug  
# Jack Russell Terrier  
# English Springer Spaniel  
# German Shepherd  
# Staffordshire Bull Terrier  
# Cavalier King Charles Spaniel  
# Golden Retriever  
# West Highland White Terrier  
# Boxer  
# Border Terrier
```

```
1 Pug  
2 Jack Russell Terrier  
3 English Springer Spaniel  
4 German Shepherd  
5 Staffordshire Bull Terrier  
6 Cavalier King Charles Spaniel  
7 Golden Retriever  
8 West Highland White Terrier  
9 Boxer  
10 Border Terrier
```

178

## Esempio lettura del file - readline()

```
with open('dog_breeds.txt', 'r') as reader:  
    print(reader.readline(5))  
    print(reader.readline(5))  
    print(reader.readline(5))  
    print(reader.readline(5))
```

# Pug

# Jack

# Russe

# ll Te

```
1 Pug  
2 Jack Russell Terrier  
3 English Springer Spaniel  
4 German Shepherd  
5 Staffordshire Bull Terrier  
6 Cavalier King Charles Spaniel  
7 Golden Retriever  
8 West Highland White Terrier  
9 Boxer  
10 Border Terrier
```

179

## Iterare su ogni riga del file

Una cosa comune da fare durante la lettura di un file è di iterare ogni riga. Per eseguire questa operazione possiamo usare il ciclo while o il ciclo for:

La funzione *print()* di Python viene fornita con un parametro chiamato 'end'. Per impostazione predefinita, il valore di questo parametro è '\n', ovvero il nuovo carattere di riga. È possibile terminare l'istruzione con qualsiasi carattere/stringa specificando questo parametro.

```
with open('dog_breeds.txt', 'r') as reader:  
    line = reader.readline()  
    while line != '': # EOF?  
        print(line, end='')  
        line = reader.readline()
```

```
with open('dog_breeds.txt', 'r') as reader:  
    for line in reader.readlines():  
        print(line, end='')
```

180

## Metodi di scrittura del file

Metodo	Descrizione
write(stringa)	Scrive nel file la stringa e ritorna il numero di caratteri (o byte) scritti.
writelines(lista)	Scrive nel file una lista in righe.

```
with open('dog_breeds.txt', 'r') as reader:
    # Note: readlines doesn't trim the line endings
    dog_breeds = reader.readlines()

with open('dog_breeds_reversed.txt', 'w') as writer:
    # alternativamente si può usare
    # writer.writelines(reversed(dog_breeds))

    # scrive le razze di cani nell'ordine inverso
    for breed in reversed(dog_breeds):
        writer.write(breed)
```

181

## Nota: apertura contemporanea di file

```
d_path = 'dog_breeds.txt'
d_r_path = 'dog_breeds_reversed.txt'
with open(d_path, 'r') as reader, open(d_r_path, 'w') as writer:
    dog_breeds = reader.readlines()
    writer.writelines(reversed(dog_breeds))
```

182

## Esercizio - File

Si assuma che un file di nome `Esami.txt` contenga gli esiti di un esame. Ogni riga contiene i seguenti dati su uno studente, separati da caratteri di spaziatura: numero di matricola e voto (un intero compreso tra 18 e 30 mentre i bocciati avranno scritto di fianco “nc”). Scrivere un programma che contenga le funzioni che dato in ingresso il file restituiscano ognuna rispettivamente:

1. Il numero di studenti promossi
2. Il numero degli studenti bocciati
3. Calcoli e stampi sullo schermo il voto medio conseguito dagli studenti promossi.

183

## Esercizio - File

Scrivere un programma in Python che permetta di analizzare un file di log “`log_macchine.txt`” che contiene le informazioni (log) sul parco macchine (Desktop PC e Server) a disposizione di una certa azienda. In particolare, ogni riga di tali file contiene le informazioni di un evento che si è verificato su una determinata macchina e contiene i seguenti campi (separati da virgola): timestamp evento, nome macchina, codice evento, versione sistema operativo. Il programma deve richiedere in input all’utente il nome della macchina di interesse e deve stampare a video tutte le righe che riguardano tale macchina. Inoltre, il programma deve stampare a video anche il numero degli eventi per la macchina di interesse registrati nel file “`log_macchine.txt`”.

Per risolvere l’esercizio, si crei un file “`log_macchine.txt`” dal contenuto a piacere che rispetti il formato definito nella consegna.

### **Esempio:**

Inserisci nome macchina di interesse:

→ `server_web`

Elenco eventi:

`1594739416, server_web, RIAVVIO ANOMALO, LINUX_5.0`

`1594738345, server_web, CRASH SISTEMA, LINUX_4.9.0`

Numero eventi per la macchina `server_web`: 2.

184

## Esercizio 1 (1/2)

È dato un file contenente una lista di persone, ad esempio il seguente:

```
Giacomo Bettini  
Simon Dahdal  
Luca Pasquali Evangelisti  
Axel Caniatti  
Alessandro Gilli
```

Per ogni riga, vengono riportati, in questo ordine, il nome e il cognome della persona. Ogni persona può avere un solo nome, ma anche più cognomi.

## Esercizio 1 (2/2)

Si crei uno script Python che legga il file di testo, e che faccia uso di una funzione

```
stampa_nome_cognome(riga)
```

che prenda in input una riga di testo e stampi a video il nome e il cognome della persona, ad esempio:

```
Nome: Giacomo, Cognome: Rossi  
Nome: Claudio, Cognome: Brambilla  
Nome: Luca, Cognome: Bianchi  
Nome: Marco, Cognome: Gialli  
Nome: Alessandro, Cognome: Verdi
```

## Esercizio 2 (1/3)

È dato un file contenente le temperature lette da un macchinario industriale, ad esempio il seguente (è scritto su due colonne, ma è un unico file):

23.56	96.85
27.91	99.42
34.28	88.61
41.75	76.91
48.62	63.23
53.79	52.89
60.04	44.73
66.11	37.12
72.98	30.32
79.54	25.56
85.33	21.73
91.26	18.09
	15.68

## Esercizio 2 (2/3)

Si crei uno script Python che legga il file di testo, e che faccia uso di una funzione

`controlla_temperatura(riga)`

che prenda in input una riga di testo e stampi a video, oltre alla temperatura letta, un messaggio secondo questi criteri:

- Temperatura maggiore di 90 gradi → Macchinario in surriscaldamento
- Temperatura compresa fra 25 e 90 → Funzionamento normale del macchinario
- Temperatura minore di 25 gradi → Macchinario spento

Se la temperatura è maggiore di 90 gradi, la temperatura e il messaggio vanno anche scritti su un file di testo dal nome `allarmi.txt`. Si effettui questa operazione mediante una funzione `scrivi_su_file(messaggio)`.

## Esercizio 2 (3/3)

Esempio di file output allarmi.txt:

91.26 Attenzione! Temperatura superiore a 90 gradi.

96.85 Attenzione! Temperatura superiore a 90 gradi.

99.42 Attenzione! Temperatura superiore a 90 gradi.

## Esercizio 3 (1/2)

È dato un file di testo contenente del testo, ad esempio:

Nel mezzo del cammin di nostra vita  
mi ritrovai per una selva oscura,  
ché la diritta via era smarrita.

Ahi quanto a dir qual era è cosa dura  
esta selva selvaggia e aspra e forte  
che nel pensier rinova la paura!

Tant' è amara che poco è più morte;  
ma per trattar del ben ch'i' vi trovai,  
dirò de l'altre cose ch'i' v'ho scorte.

## Esercizio 3 (2/2)

Si crei uno script Python che contenga le seguenti tre funzioni:

- `conta_righe(nome_file)`
  - Prende in input il percorso del file e restituisce il numero di righe del file
- `conta_parole(nome_file)`
  - Prende in input il percorso del file e restituisce il numero di parole nel file
- `copia_file(nome_file_sorgente, nome_file_destinazione)`
  - Prende in input il percorso del file sorgente e il percorso del file destinazione e, riga per riga, copia il contenuto del file sorgente nel file destinazione

Si crei poi un altro file Python che utilizza queste tre funzioni per stampare a video il numero di righe del file, il numero di parole nel file e infine copi il contenuto del file originale in un altro file (ad esempio `file_copia.txt`).

## Gestione delle eccezioni

In Python gli errori vengono riportati e gestiti usando le **eccezioni**.

Quando facevamo un'operazione aritmetica dimenticandoci di convertire una stringa in un tipo numerico l'interprete ci lanciava un errore del tipo:

```
num = input("Inserisci un numero: ")
num = num / 3
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

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



# Gestione delle eccezioni

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

In particolare, `TypeError` rappresenta il nome dell'eccezione e il messaggio a seguire ci spiega il motivo per cui tale errore si è verificato.

*Non posso dividere un oggetto di tipo stringa per un oggetto intero.*

Se un'eccezione non viene gestita, il nostro programma termina con un errore.

193

# Gestione delle eccezioni

Come possiamo gestire un'eccezione?

Attraverso il costrutto **(try/except)** definito dal linguaggio.

Per esempio se proviamo a convertire in intero una stringa non numerica otteniamo la seguente eccezione:

```
>>> n = int('ten')
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

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

194

# Gestione delle eccezioni

Per rendere il codice “safe” dobbiamo gestire l’eccezione di `ValueError` nel seguente modo:

```
>>> try:
...     n = int('dieci')
... except ValueError:
...     print('Numero non valido!')
```

195

# Gestione delle eccezioni

`try` è la parola chiave del linguaggio con cui identificare un blocco di codice che potrebbe generare un’eccezione.

Dopo il `try`, utilizziamo `except` per definire il codice da eseguire nel caso in cui si verifichi una determinata eccezione. Le istruzioni definite nel blocco di `except` vengono eseguite **se e solo se** si verifica un’eccezione. Il ramo `else` viene eseguito se non si verificano eccezioni.

```
try:
    Istruzioni
except NomeEccezione:
    Istruzioni da eseguire nel caso di eccezione
else:
    Se non si sono verificate eccezioni
```

196

# Gestione delle eccezioni

Dopo il try, si possono specificare più except ognuno con un'eccezione diversa. Infatti, potrei dover gestire diversamente problemi diversi...

Esiste anche un metodo “**meno elegante**” con cui gestire qualunque tipo di eccezione. Consiste nell'utilizzare except senza specificare il tipo dell'eccezione.

```
try:
    Istruzioni
except:
    Istruzioni
else:
    Istruzioni
```

197

## Eccezioni/errori più comuni

Eccezione	Causa dell'errore
TypeError	Una funzione o un'operazione è stata applicata a un oggetto di tipo errato. Esempio: sommare un intero a una stringa.
ValueError	Una funzione ha ricevuto in input un oggetto di tipo corretto, ma di valore improprio. Esempio: dare in input alla funzione sqrt (si trova nella libreria math), che calcola la radice quadrata, un numero negativo.
IndexError / KeyError	Si è cercato di accedere a un indice al di fuori di un iterabile (es. una lista) / a una chiave non presente in un dizionario.
FileNotFoundError	Il file al quale si vuole accedere non è stato trovato.
PermissionError	Si sta cercando di accedere a un file ma non si hanno i permessi necessari per farlo.
OSError	L'operazione che si sta cercando di eseguire ha causato un errore a livello del sistema operativo (ad esempio, FileNotFoundError e PermissionError sono sotto-eccezioni di OSError).
ZeroDivisionError	Si sta cercando di dividere una quantità per zero.

# Esercizio

Si scriva una funzione Python che realizzi la lettura “safe” di un numero intero.

La funzione deve richiedere all’utente l’inserimento di un numero intero, es. 12, e deve convertire la stringa letta tramite `input()` in un numero intero.

La funzione deve gestire le eccezioni che si possono verificare durante la conversione da stringa a intero. In caso di errore, la funzione deve continuare a richiedere il numero all’utente fino a quando avviene un inserimento corretto.