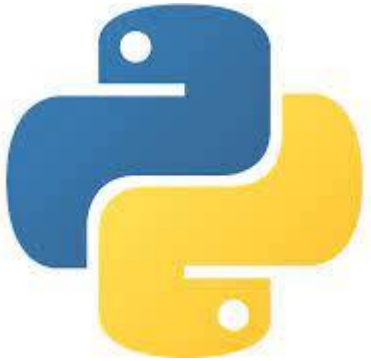


Introduzione Python

A CURA DI LINO POLO

Contesto e Importanza:

Python è attualmente uno dei linguaggi di programmazione più popolari al mondo.

Ha una sintassi chiara e la facilità di apprendimento lo rendono accessibile sia ai principianti che agli sviluppatori esperti.

Questa popolarità è sottolineata dalla vasta comunità di sviluppatori e dagli innumerevoli progetti che utilizzano Python.

Domini di uso

sviluppo web (con framework come Django e Flask), l'analisi dei dati (con librerie come Pandas e NumPy), l'intelligenza artificiale e il machine learning (con TensorFlow e PyTorch), l'automazione di task e script (con la sua sintassi concisa).

versatilità

consente agli sviluppatori di lavorare su una vasta gamma di progetti, dallo sviluppo di applicazioni web complesse alla prototipazione rapida di algoritmi di machine learning.

Comunità e Supporto

vasta documentazione ufficiale, ma anche dalla partecipazione a forum, gruppi di discussione e conferenze.

Comando pip

Il comando pip è uno strumento di gestione dei pacchetti in Python, utilizzato per installare e gestire librerie di terze parti. Il nome "pip" è un acronimo ricorsivo che sta per "Pip Installs Packages" o, più recentemente, "Pip Installs Python".

Installazione di Pacchetti:

Il comando principale di pip viene utilizzato per installare pacchetti da PyPI (Python Package Index) o da altre fonti. L'uso di base è pip install nome_pacchetto. Ad esempio, pip install requests installerà il pacchetto requests.

pip install nome_pacchetto

PyPI

PyPI sta per Python Package Index, ed è un repository online che ospita pacchetti software sviluppati e condivisi dalla comunità Python. In altre parole, PyPI è una sorta di deposito centrale di pacchetti Python che può essere utilizzato con il comando pip per installare librerie e strumenti di terze parti.

Deposito di Pacchetti Python:

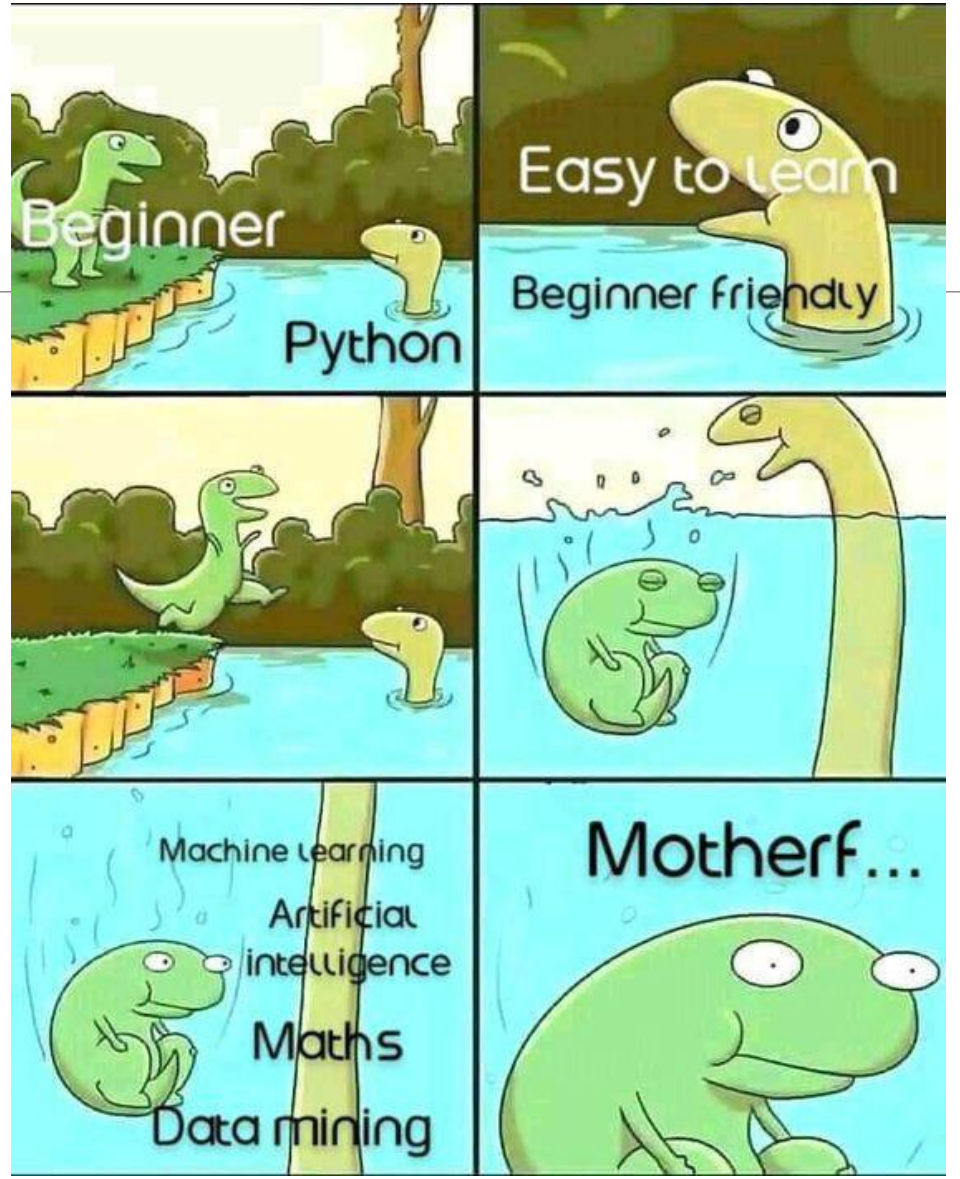
PyPI funge da deposito centrale per i pacchetti Python. Gli sviluppatori possono caricare i loro pacchetti su PyPI in modo che altri possano facilmente accedervi e installarli.

Facilita la Distribuzione e l'Installazione:

PyPI semplifica la distribuzione del software Python. Gli sviluppatori possono caricare i loro pacchetti su PyPI in modo che siano facilmente accessibili agli altri. I utenti, d'altro canto, possono utilizzare il comando pip per installare questi pacchetti con facilità.

Ricerca di Pacchetti:

PyPI offre una funzione di ricerca che consente agli sviluppatori di trovare pacchetti in base al loro nome o alla loro descrizione. Questa funzione di ricerca è spesso utilizzata per esplorare nuove librerie e strumenti disponibili


Beginner
Python
Easy to Learn
Beginner friendly
Machine Learning
Artificial intelligence
Maths
Data mining
Motherf...

Librerie Python per il Machine learning

Python permette di sviluppare funzioni molto complesse le quali, insieme alle funzioni, possono essere raccolte in librerie e condivise con altri sviluppatori, i quali potranno farne uso senza doverne riscrivere il codice.

Le librerie – termine ereditato letteralmente dall'inglese "library" – non sono una peculiarità di Python, esistono infatti per qualsiasi linguaggio di programmazione.

Le librerie di Python per il Machine learning

Come detto, Python sposa la Data science in generale e il Machine learning, attività alle quali è deputato in virtù delle tante librerie disponibili. Qui elenchiamo alcune librerie tra le più utilizzate, seguendo però il principio della loro trasversalità, a cominciare dalle funzioni matematiche che sono l'humus della Data science.

NumPy

È la libreria principale per la matematica, permette di lavorare in modo più veloce con matrici e vettori. La funzione ndarray consente di cambiare il numero delle dimensioni di un array, strutture dati ispirate proprio ai vettori e alle matrici. Tra le funzioni matematiche comprese in NumPy ce ne sono molte necessarie al Machine learning.

Riteniamo che sia una libreria fondamentale perché è la base di molte altre librerie del Python.

Pandas

Altra libreria cardinale per la Data science, l'analisi dei dati e il Machine learning, comprende funzioni per la pulizia e l'importazione di set di dati. Le strutture principali sono dataframe (di fatto tabelle con righe e colonne) e series, ossia array unidimensionali. Pandas facilita il merge e lo splitting dei dati, così come le operazioni per filtrare i dati per righe e colonne. Basata su NumPy, si integra con altre librerie orientate alla Data science.

Un pacchetto progettato per lavorare sia con i dati organizzati in database relazionali sia con dati etichettati utili al Deep learning e quindi al Machine learning.

SciPy

Libreria che contiene una collezione di algoritmi usati in ambito scientifico che rendono Python strumento paragonabile ad ambienti specifici per il calcolo e l'analisi statistica quali Mtlab o SciLab. Ogni sezione di SciPy copre argomenti peculiari del calcolo scientifico, attira il favore degli utenti perché ritenuta più intuitiva di NumPy, parere questo che ci limitiamo a riportare. Ciò che è inconfutabile è la vasta documentazione a corredo di questa libreria, implementata a sua volta su NumPy e votata al Machine learning.

PyTorch

È una libreria particolarmente nota a chi opera nel comparto della visione artificiale del riconoscimento del linguaggio naturale. Sviluppata in gran parte da Meta AI e scritta in Lula, è stata pensata per il Deep learning e le reti neurali e si sta guadagnando un proprio spazio, nonostante la presenza ingombrante (ma giustificata) di framework come TensorFlow. Dalla sua, PyTorch ha l'uso del calcolo dinamico, un vantaggio quindi nella costruzione di architetture di alto livello e, nel medesimo tempo, l'uso risulta maggiormente intuitivo.

XGBoost

La libreria XGBoost è molto usata nel Gradient Boosting, metodo che fornisce previsioni accurate soprattutto in ambienti Big data e usato nell'apprendimento supervisionato. Gli algoritmi di incremento del gradiente aiutano a ridurre gli errori di bias i quali, insieme a quelli di varianza, sono tra i problemi più stringenti in materia di apprendimento automatico. XGBoost combina modelli costruiti sui set di dati di addestramento per creare un altro modello nel quale vengono corretti gli errori riscontrati.

Scikit-Learn

Usa le capacità di algebra lineare e gli array ad alte prestazioni di NumPy per restituire modelli statistici. Sickit-Learn è basato sulla già ciatata NumPy e su SciPy e bene si adatta al Machine learning in Python, grazie alla gamma di strumenti che mette a disposizione. Al contrario, nonostante fornisca anche metodi per le implementazioni di reti neurali, ha alternative più valide nel campo del Deep learning.

Scikiit-Learn fornisce anche strumenti per le trasformazioni dei dati, elemento fondamentale nello sviluppo di modelli di previsione.

Keras

Facilita i test delle reti neurali e il loro addestramento. Keras consente di costruire modelli e di analizzare grandi set di dati. È progettata come interfaccia per altre librerie di più basso livello, tra le quali TensorFlow, Theano e il framework per l'apprendimento automatico Microsoft Cognitive Toolkit che, in Keras, sono chiamati backend.

Punta molto sulla velocità di implementazione, infatti rende possibile la creazione di modelli complessi con un numero esiguo di righe di codice. Scalabile e flessibile, è molto utilizzata anche perché permette di passare da un backend all'altro consente di distribuire i processi di Deep learning su più unità di elaborazione grafica (Gpu).

Gradio

Gradio è particolarmente usata per creare **interfacce grafiche** per i modelli di Machine learning al fine di interagire con i modelli stessi e visualizzarne i risultati.

Trova ambito di applicazione durante le fasi di **test** e di dimostrazione ma, più in generale, crea un collegamento tra gli scienziati dei dati e chi deve valutarne i risultati. Consente facilmente di creare web App e di rendere visibili i risultati del proprio lavoro anche a platee distanti e di interagire con i diversi modelli.

Permette di importare qualsiasi tipo di dato, non soltanto **tabellare**

Come commentare in Python

```
# Inizializziamo la variabile x a 0

x = 0
```

La funzione integrata type()

Ora, come fare a riconoscere i vari tipi di dato di Python? Chiaramente durante questo corso impareremo a riconoscerli e a usarli, ma come fare ciò tramite Python?

```
>>> x = 69

>>> type(x)
```

La funzione isinstance()

isinstance(object, type)

Gli Operatori Numerici che abbiamo visto sono:

addizione: +

sottrazione: -

divisione: /

moltiplicazione: *

esponente: **

quoziente: //

resto: %

Variabili e Stringhe

val = 9.81

frase = 'Una volta imparato, Python si dimostra estremamente versatile’

"e Dante scrisse: \"Nel mezzo del cammin di nostra vita...\"" 'e Dante scrisse: "Nel mezzo del cammin di nostra vita..."'

```
eggs = "Meglio un uovo oggi..."
>>> bacon = "o una gallina
domani?" >>> eggs + bacon
'Meglio un uovo oggi...o una
gallina domani?'
```

Conversioni tra tipi di dato in Python

```
>>> age = 30

>>> text = "La mia età è: "

>>> text + age

# output
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    text + age
TypeError: can only concatenate str (not "int") to str
```

Conversioni tra tipi di dato in Python

```
>>> testo + str(age)


# output

'La mia età è... 30'
```

Conversioni tra tipi di dato in Python

```
>> arance_per_sacco = "13«


>>> arance_per_sacco * 3


# output
'131313’
>>> int(arance_per_sacco) * 3
39
```

Conversioni tra tipi di dato in Python

```
>>> float(age)
30.0
```

La funzione print in Python

```
>>> print(str(33) + " trentini entrarono a Trento... passando per la funzione print!")


# output
'33 trentini entrarono a Trento... passando per la funzione print!’
>>> print("""prima linea
seconda linea
terza linea...""")
```

La funzione input in Python

La funzione input() invece ci serve per l'inserimento di dati da parte dell'utente, in fase di esecuzione.

Questo ci permette di rendere le cose estremamente interattive; ciò che inserirete tramite input() verrà passato a Python sotto forma di stringa.

Primo Programma python

```
# il tuo primo programma Python


robot_name = "Chappie"

robot_age = 1


print("Ciao! Il mio nome è " + robot_name + " e ho " + str(robot_age) + " anno")


user_name = input("Tu come ti chiami? ")

print("Ciao " + user_name + "!")


user_age = int(input("Quanti anni hai? "))

age_difference = user_age - robot_age


print(str(user_age) + " anni!? Sono " + str(age_difference) + " più di me!")

print("A presto!")
```

Valori Booleani di Python

True

False

cancello = True

oppure

cancello = False

Operatori di Comparazione in Python

OPERATORE	SIGNIFICATO
==	Uguale a
!=	Non uguale a
<	Minore di
>	Maggiore di
<=	Minore o uguale a
>=	Maggiore o uguale a

Operatori Booleani in Python: and, or, not

Operatore and

Espressione	Risultato
True AND True	True
True AND False	False
False AND True	False
False AND False	False

Operatore or

Espressione	Risultato
True OR True	True
True OR False	True
False OR True	True
False OR False	False

Operatore not

Espressione	Risultato
NOT True	False
NOT False	True

Istruzioni if, elif ed else

L'istruzione if in Python

```
age = 26

if age >= 18:
    print("Sei maggiorenne")
```


When you switch from C++ to Python
Thanks guys
pointers
;
{}
main()
++
So long, partner

L'indentazione in Python è obbligatoria

In Python, i blocchi di codice vengono delineati tramite quella che in programmazione chiamiamo **indentazione**, ovvero l'aggiunta di spazi o tab (4/8/12 ecc spazi bianchi oppure il più pythonico 1/2/3 ecc TAB) come delimitatore della nuova porzione di codice.

ATTENZIONE: a differenza di altri linguaggi di programmazione, l'indentazione in Python è **obbligatoria**, e non rispettarla comporta la comparsa di errori.


C#
PHP
Java
How dare you forget to add semicolon at the end of the statement on line 120
C#
PHP
Python
Java
IndentationError: expected an indented block after function definition on line 1

L'istruzione else in Python

```
age = 16

if age >= 18:
    print("Sei maggiorenne")
else:
    print("Sei minorenne")
```

L'istruzione elif in Python

```
age = 18

patente = False


if age >= 18 and patente == True:
    print("Puoi noleggiare una Ferrari!")
elif age >= 18 and patente == False:
    print("Fatti prima la patente!")
else:
    print("Ripassa tra qualche anno!")
```

L'istruzione pass

L'istruzione **pass** viene utilizzata quando è necessario specificare un'istruzione in un determinato punto del codice, ma non si vuole eseguire alcuna operazione. Ad esempio, può essere utilizzata come segnaposto quando si sta lavorando su una struttura di controllo del flusso (come un'istruzione if) e si desidera evitare un errore di sintassi. Ecco un esempio di come utilizzarla:

```
if x < 0:
    pass # Qui potrebbe essere inserito del codice in futuro
else:
    print("x è maggiore o uguale a 0")
```

Ciclo while, istruzioni break e continue

L'Istruzione while in Python

Il nome while loop racchiude in se la spiegazione del suo funzionamento e il perché della sua esistenza.

La parola while può infatti essere tradotta in italiano come finché o mentre, e il ciclo while ci permette proprio di eseguire un blocco di codice finché una determinata condizione è e resta True.

Facciamo subito un esempio di codice per prendere familiarità con la sintassi e chiarirci le idee:

L'Istruzione while in Python

```
counter = 0
while counter <= 10:
    print(counter)
    counter += 1
```

Cicli while Infiniti

```
counter = 0

run = True


while run == True:
    print(counter)
    counter += 1
```

L'istruzione break

```
run = True
stop = 1000
counter = 0

while run == True:
    print(counter)
    counter += 1
    if counter > stop:
        print("Sto uscendo dal loop...")
        break
Print("exit")
```

L'istruzione continue

```
run = True
skip = 5
counter = 0

while counter < 10:
    counter += 1
    if counter == skip:
        print("Sto saltando " + str(skip))
        continue
    print(counter)
```

Il Ciclo for e la funzione range

Il Ciclo for in Python

Il **for loop** viene utilizzato in maniera simile al **while loop** di cui abbiamo parlato nella lezione precedente: essendo entrambi cicli, permettono di ripetere l'esecuzione di una determinata porzione di codice più volte. Tuttavia, le differenze nei meccanismi di funzionamento ne rendono preferibile uno rispetto all'altro a seconda del contesto in cui ci si trova.

A livello logico, la differenza col ciclo **while** è che invece di eseguire il blocco di codice interessato finché la condizione di controllo resta *True*, il ciclo for esegue questa sezione di programma per un numero specifico di cicli.

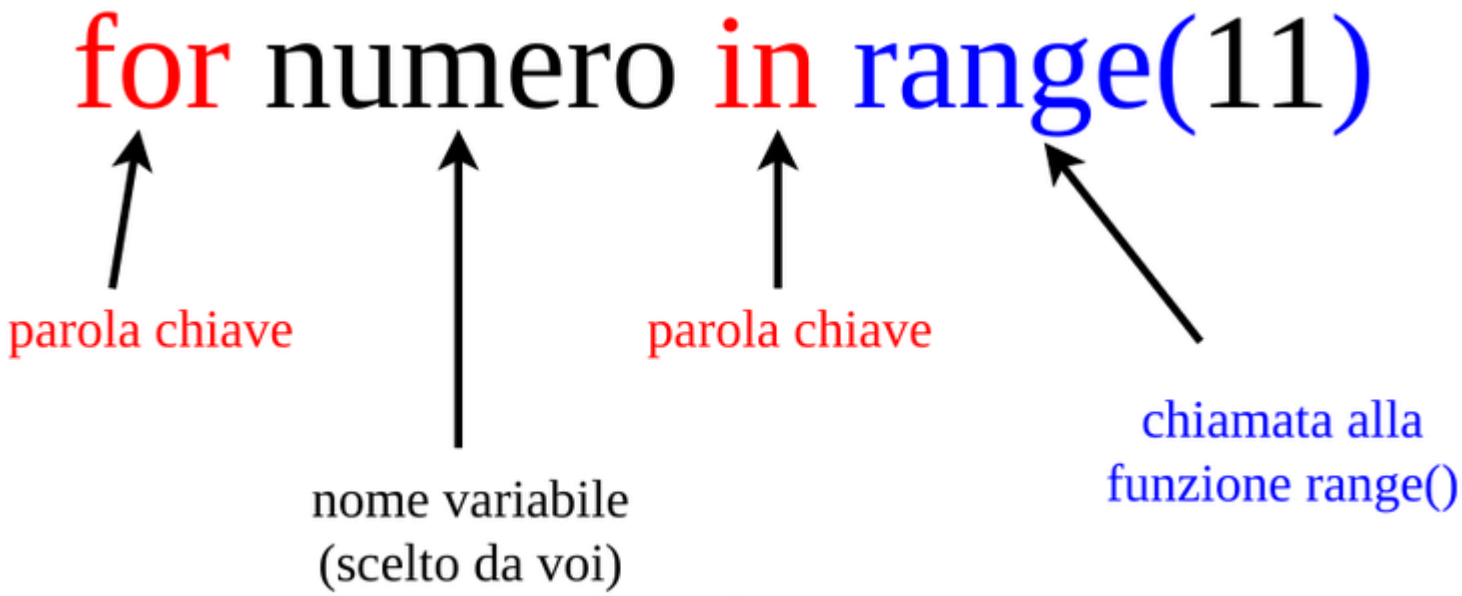
Il Ciclo for in Python

```
counter = 0

while counter <= 10:

    print(counter)

    counter += 1

----------------------------------------

for numero in range(11):

    print(numero)
```

La funzione range in Python

La funzione range() ci permette di impostare un intervallo di esecuzione tanto ampio quanto il numero che le passiamo come parametro, senza però includere il numero passato in sé. Di default, la funzione range() di Python inizia a contare a partire da 0; per comodità potete pensare a questo intervallo come a una sorta di lista, in questo caso di 11 elementi, da 0 a 10 inclusi.

La funzione range in Python



I parametri della funzione range: start, stop e step

I parametri della funzione range: start, stop e step

```
for numero in range(3, 11, 2):
    print(numero)
```

I Moduli della Standard Library

Python dispone di una collezione di moduli pronti all'uso in quella che viene chiamata la Standard Library.

Ciascun modulo è come una libreria (un insieme) di funzioni scritte da altri developer molto esperti, realizzate per svolgere compiti comuni come ad esempio la generazione di numeri casuali nel caso del modulo random o funzioni matematiche come nel caso del modulo math, interfacciamento col sistema operativo nel caso del modulo os e tanto altro ancora.

https://docs.python.org/3/library/

Alcuni esempi

- random
- math
- datetime
- platform
- tkinter
- turtle
- os
- shutil
- re
- zipfile
- json

Come importare un modulo

Ci sono tre diversi modi per importare un modulo nei nostri script di Python.

Il primo consiste semplicemente nel chiamare il modulo tramite il comando import. Con questo tipo di import potete accedere alle funzioni e alle classi del modulo nel vostro codice utilizzando la dot notation: nome_modulo.nome_oggetto.

import math

Come importare un modulo

Il secondo consiste nell'importare solo specifiche classi, variabili e funzioni dal modulo utilizzando **from** nome_modulo **import** nome_oggetto: in tal caso possiamo utilizzare ciò che abbiamo importato chiamando gli oggetti direttamente per nome senza far ricorso alla dot notation.

from math import sqrt, exp

Come importare un modulo

Il terzo consiste nel richiamare tutte le funzioni, le classi e le variabili incluse nel modulo tramite l'asterisco, e anche in questo caso non sarà necessario riscrivere il nome del modulo.

```
from math import *
```

Il modulo random

Questo modulo fornisce una serie di funzioni per generare numeri pseudo-casuali. I numeri ottenuti con random possono essere utili in molte situazioni, per esempio quando è necessario simulare eventi, testare algoritmi o selezionare elementi casuali da una lista.

random.random()

Il metodo randint(a, b)

```
import random
for numero in range(10):
    val = random.randint(1, 20)
    print(val)
```

Il modulo random

Il metodo randrange(start, stop[, step])

Genera solo numeri interi pari

random.randrange(0, 10, 2)

Genera solo numeri interi dispari

random.randrange(1, 10, 2)

Il modulo random

Il metodo choice(seq)

Seleziona un elemento casuale dalla sequenza *seq*, che può essere una lista, una tupla o una stringa.

```
lista = ['Io', 'Ganimede', 'Callisto', 'Europa']


random.choice(lista)


# output

'Europa'
```

Il modulo math

Questo modulo è utilizzato per eseguire operazioni matematiche e la maggioranza delle sue funzioni restituiscono valori di tipo float. Vediamo alcune delle funzioni più utilizzate di math:

Il metodo sqrt(x)

Restituisce la radice quadrata di x:

math.sqrt(25)

Il metodo pow(x, y)

Restituisce x elevato alla potenza di y:

```
math.pow(10, 3)


# output
1000.0
```

Il metodo exp(x)

Restituisce l’esponenziale di x, cioé e elevato x:

```
math.exp(x)

# output
7.38905609893065
```

I metodi ceil(x) e floor(x)

La prima restituisce l’arrotondamento per eccesso di x (il più piccolo numero intero maggiore o uguale a x), la seconda restituisce l’arrotondamento per difetto di x (il più grande numero intero minore o uguale a x).

```
math.ceil(33.23)


# output

34


math.floor(23.532)


# output

23
```

I metodi sin(x), cos(x) e tan(x)

Restituiscono rispettivamente il seno, il coseno e la tangente di x in radianti.

```
math.sin(1.5707963267948966)
# output
1.0
math.cos(0)
# output
1.0
math.tan(0.7853981633974483)
# output
0.9999999999999999
```

I metodi degrees(x) e radians(x)

Convertono l'angolo x da gradi a radianti e viceversa.

```
math.degrees(1.0471975511965976)
# output
59.99999999999999
math.radians(45)
# output
0.7853981633974483
```

Il modulo datetime

In Python il tempo non rappresenta un tipo specifico di dato: questo modulo **permette di creare oggetti per rappresentare data e ora** e fornisce tutta una serie di metodi per effettuare con esse varie operazioni, come convertire gli oggetti in stringhe o calcolare durate. Alcune delle classi più utilizzate di **datetime** sono le seguenti:

La classe date()

Restituisce un oggetto date che rappresenta una data specificata dall'utente. Come parametri opzionali accetta l’anno, il mese e il giorno:

```
d = datetime.date(2023, 11, 16)

print(f"{type(d)}: {d}")


# output

<class 'datetime.date'>: 2027-01-01
```

La classe time()

Restituisce un oggetto time che rappresenta un orario specificato dall'utente. Come parametri opzionali accetta l’ora, il minuto, il secondo e il microsecondo:

```
t = datetime.time(8, 57, 15)
print(f"{type(t)}: {t}")

# output
<class 'datetime.time'>: 08:57:15
```

La classe datetime()

Restituisce un oggetto datetime che rappresenta una data e un orario specificati dall'utente. Come parametri opzionali accetta l’anno, il mese, il giorno, l’ora, il minuto, il secondo e il microsecondo:

```
dt = datetime.datetime(2025, 5, 25, 1, 1, 1)

print(f"{type(dt)}: {dt}")


# output

<class 'datetime.datetime'>: 2025-05-25 01:01:01
```

La classe timedelta()

Restituisce un oggetto timedelta che rappresenta una durata di tempo. Come parametri opzionali accetta settimane, giorni, ore, minuti, secondi, millisecondi e microsecondi.

```
td = datetime.timedelta(days=5, hours=7, minutes=30, seconds=2)

print(f"{type(td)}: {td}")


# output

<class 'datetime.timedelta'>: 5 days, 7:30:02
```

Il modulo platform

Questo modulo fornisce una serie di metodi per ottenere informazioni sul sistema operativo, la macchina e l'ambiente in cui il codice Python viene eseguito. Queste informazioni possono essere utili per personalizzare il comportamento del codice in base all'ambiente di esecuzione.

```
platform.system()

# output

'Windows'

platform.machine()

# output

'AMD64'

platform.processor()

# output

'Intel64 Family 6 Model 141 Stepping 1, GenuineIntel'

platform.architecture()

# output

('64bit', 'WindowsPE')

platform.python_version()

# output

'3.10.3'
```

Il modulo tkinter

Questo modulo fornisce un’interfaccia di programmazione per creare applicazioni grafiche multipiattaforma. La libreria contiene una serie di widget come pulsanti, etichette, caselle di testo, menu e finestre, che possono essere utilizzati per creare interfacce utente interattive.

È possibile creare finestre e widget, impostare il loro aspetto e la loro posizione, gestire gli eventi generati dall'utente e molto altro ancora. Creiamo come esempio una finestra che contiene un semplice pulsante cliccabile:

```
import tkinter as tk


root = tk.Tk()

root.geometry("400x400")

button = tk.Button(root, text='Cliccami!')

button.pack()
```

Il modulo turtle

Questo modulo permette di creare immagini e disegni grafici utilizzando una "tartaruga" virtuale controllabile tramite una serie di comandi da inserire nel codice. Ad esempio, se avviate questo codice vedrete che la tartaruga disegnerà un quadrato:

```
import turtle


t = turtle.Turtle()


for i in range(4):
    t.forward(100)
    t.left(90)


turtle.done()
```

Il modulo os

Viene utilizzato per interagire con il sistema operativo. Le sue funzionalità sono tantissime, ma quelle che incontrerete più spesso riguardano la navigazione nelle cartelle del filesystem. Molti metodi di os accettano come parametro un percorso (in inglese path):

Il metodo listdir(path)

Questo metodo restituisce una lista contenente i nomi di tutti i file e tutte le cartelle della directory specificata dal parametro path, in ordine arbitrario:

```
os.listdir('C:\\Users\\Username\\Documents')


# output

['Lavoro', 'Testi Canzoni', 'Pdf', 'Nuova Cartella']
```

Il modulo shutil

Questo modulo (abbreviazione di **Shell Utility**) fornisce un insieme di funzioni per la **gestione del filesystem** e semplifica molte operazioni come copiare, spostare o eliminare file o directory, archiviare in formato ZIP, ecc.

Il modulo re

Questo modulo permette di compiere vari tipi di operazioni con le espressioni regolari (re è l'acronimo di regular expression), ovvero sequenze di caratteri utilizzate per cercare, sostituire e manipolare stringhe di testo.

Tutti i metodi di re che vedremo accettano come parametro opzionale anche flags, che permette specificare comportamenti aggiuntivi per la funzione, ad esempio re.IGNORECASE, che esegue la ricerca senza considerare maiuscole e minuscole, e re.MULTILINE, che esegue la ricerca su più righe del testo.

Il metodo search(pattern, string)

Questo metodo cerca la stringa pattern all’interno della stringa string e restituisce un oggetto match corrispondente alla prima occorrenza trovata, dove span è una tupla contenete gli indici della posizione iniziale e finale della corrispondenza nella stringa.

```
string = "All work and no play makes Jack a dull boy"

pattern = "l"


re.search(pattern, string)


# output

<re.Match object; span=(1, 2), match='l'>
```

Il metodo findall(pattern, string)

Questo metodo cerca la stringa pattern all’interno della stringa string e restituisce una lista di stringhe.

```
string = "All work and no play makes Jack a dull boy"
pattern = "jack"

re.findall(pattern, string, re.IGNORECASE)

# output
['Jack']
```

Il metodo sub(pattern, repl, string)

Questo metodo sostituisce tutte le occorrenze della stringa pattern all’interno della stringa string con la stringa repl.

```
stringa = "La mia email è mia.mail@gmail.com."

pattern = r"\b\w+\.\w+@\w+\.\w+\b"

nuova_stringa = re.sub(pattern, "EMAIL", stringa)

print(nuova_stringa)

# output

La mia email è EMAIL.
```

Il modulo zipfile

Questo modulo fornisce diverse funzionalità per la **creazione, la lettura e la modifica di archivi ZIP**, che sono file compressi contenenti uno o più file o directory. Tramite **zipfile** possiamo estrarre i file contenuti nell file .zip e aggiungerli o rimuoverli senza dover decomprimere e ricreare l'intero archivio.

Il modulo json

Questo modulo serve per codificare e decodificare i dati in formato JSON (JavaScript Object Notation): si tratta di un formato molto comune utilizzato soprattutto nella comunicazione tra client e server e come file di configurazione (anche su VSCode). La struttura di un oggetto JSON è simile a quella di un dizionario: i dati sono strutturati come coppie chiave: valore (dette proprietà) circondate da parentesi graffe e separate tra loro da una virgola.

Per aprire i file JSON si utilizza l'istruzione with.

La funzione load(fp)

Accetta come argomento un file da cui caricare i dati JSON e restituisce un oggetto Python corrispondente in base alla tabella di conversione:

```
with open('pianeti.json', 'r') as file:
    data = json.load(file)
print(data)

# output
{'pianeta': 'Giove', 'massa terrestre': 317.83, 'satelliti': 95}
```

La funzione loads(s)

Accetta come argomento una stringa s contenete dei dati JSON e la converte in un oggetto Python in base alla tabella di conversione:

```
elemento = '{"nome": "Cloro", "simbolo": "Cl", "numero atomico": 17}'
data = json.loads(elemento)
print(data)

# output
{'nome': 'Cloro', 'simbolo': 'Cl', 'numero atomico': 17}
```

La funzione dump(obj, fp)

Accetta come argomento obj un oggetto da convertire in formato JSON e come argomento fp un file su cui scrivere i dati:

```
data = {'nome': 'Silvia', 'età': 30, 'città': 'Milano'}

with open('data.json', 'w') as file:
    json.dump(data, file)
```

La funzione dumps(obj)

Accetta un oggetto (obj) da convertire in formato JSON e restituisce una stringa contenente i dati:

```
data = {'nome': 'Mario', 'età': 45, 'città': 'Roma'}
json_data = json.dumps(data)
print(json_data)

# output
{"nome": "Mario", "et\u00e0": 45, "citt\u00e0": "Roma"}
```

Come creare una calcolatrice in Python

Abbiamo finora appreso come usare varie funzioni e istruzioni proprie del linguaggio. Tra tutto quanto visto finora, cosa utilizzeremo in questa lezione - tutorial?

- Gli operatori numerici di Python e le variabili
- La funzione print() e la funzione input()
- Le istruzioni di controllo if, elif ed else
- Alcune funzioni di conversione
- Il ciclo while e le istruzioni break e continue

Le Funzioni in Python

Riutilizzo del codice

Uno dei concetti comuni nel mondo della programmazione è il concetto di riutilizzo del codice. Generalmente, più un programma diventa complesso e strutturato, più saranno le righe di codice che lo compongono. Affinché sia possibile tenere questo codice quanto più leggero possibile, uniforme e semplice da mantenere, è molto importante evitare ripetizioni superflue.

Finora abbiamo visto alcuni esempi di riutilizzo del codice: abbiamo imparato a usare i cicli di controllo, i moduli della Standard Library e alcune funzioni integrate come la funzione print(), usata più volte e in parti diverse dei nostri programmi.

In aggiunta alle funzioni integrate offerte da Python, potete scrivere le vostre funzioni per fargli fare ciò che desiderate: in questa lezione imparerete a fare proprio questo.

Ciò risulta molto importante perché ci permetterà come vedremo, di strutturare i nostri programmi in maniera efficiente ed elegante.

Le funzioni in Python

L'idea dietro al concetto di funzione è quella di assegnare una porzione di codice e delle variabili chiamate parametri ad un nome, e mediante questo nome che funge da collegamento richiamare tutto il codice associato.

Per certi versi il concetto è simile a quello di variabile che abbiamo visto nella lezione numero 3, solo che in questo caso possiamo richiamare non solo dei dati ma anche delle istruzioni.

E così come possiamo creare dei semplici programmi salvandoli all'interno di file con estensione .py, per eseguirli tramite terminale, possiamo usare le funzioni per creare dei mini programmi ed eseguirli dall'interno del codice stesso.

Le funzioni sono quindi fantastiche per fare in modo di ridurre il numero di volte che un pezzo di codice deve essere scritto in un programma e per mantenere il nostro software ordinato.

La parola chiave def

Per definire una funzione utilizziamo la parola chiave def seguita dal nome che vogliamo dare a questa sezione del nostro programma, quindi a questa funzione. Come nel caso delle variabili, cercate di usare dei nomi che siano rappresentativi dello scopo della funzione.

Il nome viene seguito da una coppia di parentesi, e all'interno delle parentesi metteremo i parametri, ovvero delle variabili necessarie al funzionamento della funzione. Non tutte le funzioni avranno bisogno di parametri: in questo primo esempio lasceremo le parentesi vuote.

```
def say_my_name():
    name = input("Come ti chiami? ")
    print("Il tuo nome è: ", name)
```

Seguendo la logica di Python, il codice inserito all'interno della funzione deve essere indentato. All'interno della funzione potete aggiungere qualsiasi istruzione riteniate opportuna.

Per eseguire la funzione dobbiamo richiamarla, e per fare una chiamata alla funzione possiamo semplicemente utilizzare il suo nome seguito dalle parentesi

```
# definizione della funzione

def say_my_name():

    name = input("Come ti chiami? ")

    print("Il tuo nome è: ", str(name))


# chiamata della funzione

say_my_name()
```

Facciamo ora un secondo esempio in cui utilizzeremo i parametri che abbiamo prima accennato. Definiamo una funzione addizione a cui passeremo due numeri e che ci restituirà la somma tra i due:

```
def addizione(a, b):
    print("Questa è la funzione addizione.")
    print("Fornisce la somma di due numeri passati come parametri.")
    risultato = a + b
    print("Il risultato dell'addizione è " + str(risultato))
```

Le docstring di Python

Una docstring è una stringa di testo che viene inserita all'inizio di una funzione, di una classe o di un modulo e che serve a documentare il codice. A differenza dei commenti, le docstring sono considerate parte del codice e vengono utilizzate per generare la documentazione automaticamente: questo significa che possono essere lette da strumenti di documentazione come Sphinx e possono essere utilizzate per generare documentazione HTML e PDF. Le docstring si scrivono inserendo il testo all'interno di stringhe multiriga definite con tripli apici """ """:

```
def somma(a, b):
    """
    Questa funzione prende due argomenti, a e b, e restituisce la loro somma.
    """
    return a + b
```

In questo caso, la docstring descrive lo scopo della funzione e il valore che viene restituito. Possono essere utilizzate anche per documentare i parametri di una funzione, ad esempio:

```
def saluta(nome):
    """
    Questa funzione prende un argomento, nome, e restituisce una stringa di saluto.

    :param nome: il nome della persona da salutare
    :return: una stringa di saluto
    """
    return "Ciao, " + nome + "!"
```

Quando si utilizza una funzione che ha una docstring in Visual Studio Code, è possibile visualizzarla attraverso la funzionalità IntelliSense dell'editor: questo significa che quando si digita il nome della funzione, verrà mostrato un suggerimento che include la docstring della funzione, compresi i parametri e il valore che viene restituito. Vediamo un esempio:

```
def calcola_media(valori):
    """
    Calcola la media dei valori nella lista fornita.

    Args:
        valori (list): una lista di numeri

    Returns:
        float: la media dei numeri nella lista
    """
    somma = 0
    for valore in valori:
        somma += valore
    media = somma / len(valori)
    return media
```

Parametri opzionali e valori di default

Talvolta ci troveremo in una situazione in cui è necessario avere dei parametri opzionali alle nostre funzioni, ovvero fare in modo che queste funzionino con o senza l'aggiunta di valori in questi specifici parametri. Per rendere questo possibile, a questi parametri è associato sempre un valore di default.

Immaginate che stiate andando a comprare un laptop nuovo. Due caratteristiche da tenere sott'occhio sono sicuramente RAM e modello della CPU, che quindi sceglieremo. Come vi sarà capitato, vi viene chiesto se volete acquistare anche un pacchetto antivirus, e voi potete scegliere se lo volete oppure meno.

Proviamo a portare questo esempio in una funzione:

```
def laptop_nuovo(ram, cpu, antivirus=False):
    print("Il nuovo laptop avrà le seguenti caratteristiche:")
    print("ram: " + ram)
    print("cpu: " + cpu)
    if antivirus == True:
        print("Hai comprato anche un antivirus!")
```

Diversamente se vogliamo l'antivirus ci basta passare True come valore per il parametro antivirus:

```
>>> laptop_nuovo("16gb", "i7", antivirus=True)
```

Il nuovo laptop avrà le seguenti caratteristiche:

ram: 16gb

cpu: i7

Hai comprato anche un antivirus!

L'istruzione return in Python

```
def addizione(a, b):
    risultato = a + b
    return risultato
```

Il tipo nullo di Python: NoneType

Se non definiamo un valore da restituire tramite return, la funzione restituirà il tipo di dato nullo o null di Python, il None Type. Modifichiamo il codice scritto in precedenza e facciamo una prova

```
def addizione(a, b):
    risultato = a + b
risultato = addizione(3, 6)
print(risultato)
print(type(risultato))


# output
None
<class 'NoneType'>
```

Variabili Globali e Variabili Locali

In questa lezione approfondiremo il discorso su variabili e funzioni parlando di variabili globali e variabili locali.

Finora abbiamo detto, semplificando il tutto a fini didattici, che possiamo immaginare le variabili come delle scatole in cui depositiamo dei valori. Possiamo quindi richiamare e modificare questo valore utilizzando il nome della variabile.

```
x = 3

x += 1

x

4
```

Abbiamo inoltre parlato di funzioni, e abbiamo visto che anche qui possiamo e siamo incoraggiati ad utilizzare le variabili: si tratta di uno degli elementi indispensabili di Python e della programmazione in generale. Ma fino a dove ci possiamo spingere nell'utilizzo delle variabili? Fino a che punto è davvero possibile utilizzarle e modificarle nei nostri programmi?

Per rispondere a queste domande andremo ora ad approfondire il tema parlando di Variabili Globali e Variabili Locali. Partiamo da un esempio:

```
x = 15

def funzione_esempio():
    return x

print(funzione_esempio())
15
```

```
x = 15
def funzione_esempio():
    x += 2
    return x
print(funzione_esempio())
# output in Python 3.11
UnboundLocalError: cannot access local variable 'x' where it is not associated with a value

# in versioni precedenti di Python, il messaggio d'errore mostrato era
UnboundLocalError: local variable 'x' referenced before assignment
```

Local Scope e Global Scope

La chiave di lettura di questo comportamento apparentemente bizzarro è la seguente, prestate attenzione: in Python il codice e quindi anche le variabili, possono essere "salvati" in due ambienti diversi chiamati local scope e global scope, traducibili sostanzialmente come ambito di visibilità locale e globale

Potete pensare a questi due ambienti come a dei contenitori distinti in cui vengono definite e assegnate le variabili, un contenitore globale e un contenitore locale. Quando uno di questi contenitori viene distrutto, quindi quando uno di questi ambiti viene distrutto, lo stesso accade per i valori delle variabili in esso salvate che vengono quindi dimenticati.

Variabili Locali e Variabili Globali

Un ambito locale viene creato ogni volta che una funzione viene chiamata, e distrutto dopo che la funzione restituisce un valore con return.

È come se ogni volta che una funzione viene processata, Python le fornisse un contenitore e le dicesse: bene, metti le tue variabili e il tuo codice quà dentro. Possono quindi esistere tanti Local Scope quante funzioni abbiamo in esecuzione. Le variabili dichiarate all'interno di qualsiasi funzione, quindi dell'Ambito Locale della funzione, sono chiamate variabili locali.

Dall'altro lato invece, esiste e può esistere un unico Ambiente Globale, che viene creato automaticamente da Python all'avvio del programma e distrutto alla chiusura del programma. È l'ambito principale e tutte le variabili che vengono dichiarate qui, quindi all'esterno di una funzione, sono chiamate proprio variabili globali, e restano pertanto in vita dall'avvio del programma "principale" fino alla sua chiusura.

Variabili Locali e Variabili Globali

È possibile accedere alle variabili globali da qualsiasi parte del programma, mentre è possibile accedere alle variabili locali solamente dall'ambito locale della funzione in cui sono contenute.

Nel nostro esempio, x è proprio una variabile globale, e otteniamo un errore perché seppur è vero che da un local scope si puó accedere alle variabili del global scope, il loro utilizzo all'interno della funzione è limitato.

L'istruzione global

Per poter modificare il valore di una variabile globale dall'interno di una funzione, come abbiamo provato a fare con la nostra x, dobbiamo prima dichiarare alla funzione le nostre intenzioni mediante l'istruzione global. Modifichiamo l'esempio precedente:

```
x = 15


def funzione_esempio():
    global x
    x += 2
    return x



print(funzione_esempio())
```

17

Ed ecco che ora ci è possibile modificare il valore della variabile x, perché Python sa che ci stiamo riferendo alla x presente nel global scope!

Un altro modo per poter utilizzare il valore di una variabile globale in maniera più complicata all'interno di una funzione è creando una variabile locale a cui assegnamo il valore della variabile globale. Potremo quindi poi modificare la nostra nuova variabile locale, aggirando così la limitazione.

```
x = 15

def funzione_esempio():
    y = x
    y += 2
    return y

print(funzione_esempio())
17
```

Come accennato inoltre, al contrario delle globali, è possibile accedere alle variabili locali solamente dall'ambito locale della funzione in cui sono contenute, quindi non è possibile accedervi dal global scope o dal local scope di un'altra funzione e otterremmo un errore qualora provassimo a forzare questa regola. Ad esempio:

```
def mia_funzione():
    val = 24
    print(val)
new_val = val + 6
# output
NameError: name 'val' is not defined
```

Potete utilizzare le variabili di una funzione solo se queste ci vengono passate dal return della funzione. Ad esempio, facciamo una piccola modifica:

```
def mia_funzione():
    val = 24
    return val


new_val = mia_funzione() + 6
print(new_val)


# output
30
```

Local e Global Scope: quanto sono importanti?

È bene imparare la distinzione tra local e global scope perché si tratta di concetti estremamente importanti. L'esistenza degli ambiti è utilissima ad esempio per ridurre il numero di bug e aumentare la robustezza dei programmi.

Pensate a che caos se in un programma complesso tutte le variabili fossero globali, e per colpa di un errore in una della tante funzioni del programma i valori delle variabili fossero perennemente sballati, causando un crash sistematico. Inoltre restando in memoria fino alla fine del programma, in programmi complessi le variabili globali sono sinonimo di pesantezza e spreco di risorse.

L'utilizzo di Variabili Locali è quindi estremamente incoraggiato, e quello di Variabili Globali decisamente SCONSIGLIATO!

Le Liste

Le Liste in Python

Il tipo di dato Lista di Python, come il nome suggerisce, ci permette di raggruppare più elementi tra di loro.

Le liste in Python sono molto potenti, possono infatti contenere al loro interno diversi tipi di dato anche tutti assieme e i vari elementi vengono ordinati in base ad un indice proprio della lista, in modo da semplificarne l'accesso.

Creiamo ora una variabile my_list ed assegniamole una lista che contenga al suo interno dati di tipo integer, float e string. Per dichiarare una lista usiamo una coppia di parentesi quadre, e i vari elementi contenuti sono separati da una virgola

```
>>> my_list = [9.81, "pasta", 22, 44, 3.14]
>>> type(my_list)


# output
<class 'list'>
```

Le liste sono così versatili che volendo possono contenere anche altre liste:

```
>>> new_list = ["asd", "poker", "luna", my_list]
>>> new_list

# output
['asd', 'poker', 'luna', [9.81, 'pasta', 22, 44, 3.14]]
```

Gli indici del tipo di dato list in Python

Abbiamo detto che gli elementi sono ordinati in base ad un indice proprio di ciascuna Lista, e ciò significa che a ciascun elemento è associato un numero che ne rappresenta la posizione nell'elenco: comprendere ciò è fondamentale perché se vogliamo accedere ad un elemento specifico avremo bisogno dell'indice corrispondente nella lista.

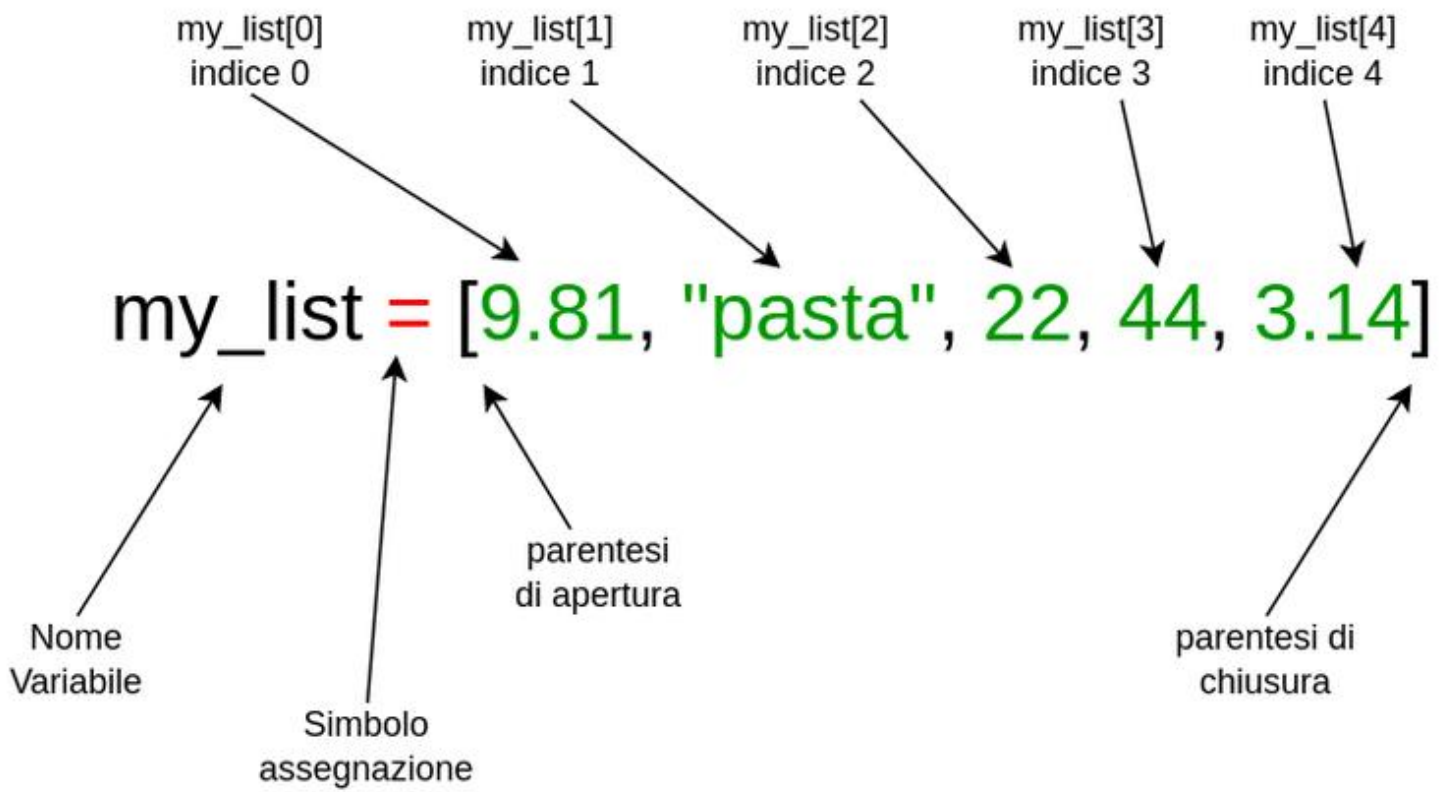
NOTA FONDAMENTALE: le liste iniziano ad indice 0.

RIPETO: il primo indice delle liste è 0.

Per richiamare "pasta" dovremo quindi usare l'indice 1:

```
my_list[1]
```

```
'pasta'
```


my_list[0]
indice 0
my_list[1]
indice 1
my_list[2]
indice 2
my_list[3]
indice 3
my_list[4]
indice 4
my_list = [9.81, "pasta", 22, 44, 3.14]
Nome
Variabile
Simbolo
assegnazione
parentesi
di apertura
parentesi di
chiusura

Talvolta può dimostrarsi utile accedere agli elementi della lista partendo dalla fine: in questo caso possiamo usare indici negativi, partendo da -1:

```
my_list[-1]
```

```
3.14
```

Facciamo ora un esempio partendo da una lista di numeri primi.

Per ottenere un insieme di elementi contenuti in una lista, possiamo usare lo slicing, che si traduce letteralmente come affettare.

Questo ci permette di ottenere una "sottolista" di elementi a partire da due indici, e volendo possiamo assegnare questa nuova lista a una nuova variabile:

```
primi = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

primi[4:10]

# output

[11, 13, 17, 19, 23, 29]


fetta = primi[4:10]

fetta

# output

[11, 13, 17, 19, 23, 29]
```

Possiamo impostare questo indice variabile anche in modo da farlo iniziare, o terminare, con l'inizio oppure la fine della lista, utilizzando i due punti :.

Per ottenere tutti i numeri dal 5o all'ultimo nella nostra lista possiamo fare:

```
>>> primi[4:]
[11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Liste e cicli for

Le liste sono davvero ovunque e vi capiterà di introntrarle spessisimo, forse anche più di altri tipi di dato! È quindi fondamentale tenere a mente che per navigare il contenuto di una lista possiamo usare ad esempio i cicli for:

```
primi = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
match = 11

for el in primi:
    if el == match:
        output = str(el) + " == " + str(match)
        print("Match! " + output)
    else:
        print(element)
```

I Metodi delle Liste

```
>>> spesa = ["riso", "pollo", "verdura"]
>>> spesa += "sapone«
# output
['riso', 'pollo', 'verdura', 's', 'a', 'p', 'o', 'n', 'e’]
>>> spesa = ["riso", "pollo", "verdura"]
>>> spesa += 3.14

# output
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object is not iterable
```

Per lavorare con le liste di Python usiamo alcune specifiche istruzioni e delle funzioni "speciali" chiamate metodi. Senza scendere troppo nei dettagli avanzati, la particolarità dei metodi è che vengono richiamati su dati di un certo tipo.

Aggiungere elementi con append() ed extend()

Volendo aggiungere metodi singoli elementi ad una lista usiamo il metodo append():

```
>>> spesa = ["riso", "pollo", "verdura"]
>>> spesa.append("sapone")
>>> spesa


# output
['riso', 'pollo', 'verdura', 'sapone']
```

Mentre per aggiungere elementi ad una lista da una seconda lista usiamo il metodo extend():

```
>>> spesa_extra = ["camicia", "copri volante"]
>>> spesa.extend(spesa_extra)
>>> spesa

# output
['riso', 'pollo', 'verdura', 'sapone', 'camicia', 'copri volante']
```

Ordinare elementi tramite il metodo sort()

Il termine sort() è traducibile in Italiano come ordinare, o classificare.

Questo metodo ci consente di ordinare i valori all'interno di una lista mediante ordine alfabetico o numerico:

```
>>> alfabeto = ["z", "u", "a", "c", "v"]

>>> alfabeto.sort()
```

Il metodo index(): come ottenere l'indice di un elemento

Gli indici delle liste sono fondamentali perché identificano la posizione degli elementi al loro interno.

Per ottenere l'indice di uno specifico elemento possiamo usare il metodo index().

```
>>> numeri = [32, 1, 15, 2, 22, 23, 56, 88]

>>> numeri.index(22)
```

Sostituire ed eliminare elementi tramite indice

Possiamo sostituire un elemento passando un nuovo valore all'indice dell'elemento che intendiamo sostituire, oppure cancellarlo:

```
>>> spesa = ["riso", "pollo", "verdura"]

>>> spesa[1] = "tacchino"

>>> spesa


# output

['riso', 'tacchino', 'verdura', 'sapone']
```

Rimuovere elementi con pop() e remove()

Infine, possiamo rimuovere elementi dalle liste usando anche i metodi pop e remove.

Con pop() rimuoviamo l'ultimo elemento della lista, mentre con remove() possiamo rimuovere elementi a partire dal loro valore:

```
>>> new_list = [True, None, "poker", 4.20, 1945]
>>> new_list
# output
[True, None, 'poker', 4.2, 1945]
################################
>>> new_list.pop()
1945
>>> new_list
# output
[True, None, 'poker', 4.2]
################################
>>> new_list.remove(None)
>>> new_list
[True, 'poker', 4.2]
```

Metodi delle Stringhe

In questa lezione approfondiremo la nostra conoscenza delle stringhe in Python introducendo alcuni metodi molti utili propri di questo tipo di dato e facendo alcune analogie col tipo di dato lista, scoprendo più di qualche punto in comune tra i due!

Finora quando abbiamo dovuto concatenare più stringhe tra di loro, abbiamo utilizzato l'operatore +:

```
>>> nome = "Jack"

>>> "Ciao " + nome

'Ciao Jack'
```

Per poter unire dei valori numerici inoltre, abbiamo sempre dovuto utilizzare la funzione di supporto str(), che ci permette di ottenere la rappresentazione in stringa:

```
numero = 218

>>> "Ciao " + nome + " il tuo posto è il n" + str(numero)

'Ciao Jack il tuo posto è il n218'
```

Questo può portare a confusione o alla creazione di righe di codice particolarmente lunghe e difficili da leggere. In alternativa possiamo usare un sistema di formattazione avanzata chiamato Formatted String Literal, o f-string.

Con le f-string valori, variabili o espressioni possono essere passati all'interno di parentesi graffe.

```
>>> f"Ciao {nome}, il tuo posto è il n{numero}! Benvenuto!"
'Ciao Jack,  il tuo posto è il n18! Benvenuto!'
```

```
>>> z = 5
>>> f"Il quadrato di {z} è {z * z}"
'Il quadrato di 5 è 25'
```

Metodi per Stringhe

Come abbiamo detto nelle lezioni precedenti del corso, i metodi sono delle "funzioni speciali" proprie nel nostro caso dei tipi

di dato. Sono proprietà di oggetti di una determinata tipologia.

Anche il tipo di dato stringa dispone di metodi propri molto comodi per permetterci di eseguire svariate comode azioni.

I metodi startswith() e endswith()

Con startswith() e endswith() possiamo controllare se una data stringa inizi o finisca con un determinato carattere o insieme di caratteri. Questi metodi restituiscono valori booleani, quindi ricordiamolo, True o False.

```
>>> messaggio = "Fate il vostro gioco"

>>> messaggio.startswith("Fate")

True

>>> messaggio.startswith("F")

True

>>> messaggio.starstwith("x")

False

>>> messaggio.endswith("gioco")

True

>>> messaggio.endswith("gioc")

False
```

I metodi isalnum(), isalpha(), isdecimal() e isspace()

Capiterà spesso che dobbiate verificare la tipologia di caratteri contenuti in una stringa, e i metodi isalnum()isalpha(), isdecimal() e isspace() ci permettono di verificare proprio ciò per la gran parte dei casi più comuni, restituendoci True se la condizione è veritiera:

metodo isalnum(): solo caratteri alfanumerici

metodo isalpha(): solo caratteri alfabetici

metodo isdecimal(): solo numeri

metodo isspace(): tutti i caratteri sono spazi bianchi

```
spam = "asd123"

eggs = "999"

bacon = "   "

monty = "poker "

>>> spam.isalnum()

True

>>> spam.isalpha()

False

>>> eggs.isdecimal()

True

>>> eggs.isalnum()

True

>>> monty.isspace()

False

>>> bacon.isspace()

True
```

I metodi upper() e lower()

I metodi upper() e lower() ci permettono di ottenere la versione maiuscola o minuscola di una stringa. Tenete però a mente che questi metodi non modificano il valore contenuto in una determinata variabile: per fare ciò sarà necessario assegnare un nuovo valore alla stessa variabile.

```
>> name = "Alice"
>>> name.lower()
'alice'
>>> name.upper()
'ALICE'
# il valore di name non è cambiato
>>> name
'Alice'
>>> name = name.upper()
# il valore di name è ora stato cambiato
>>> name
'ALICE'
```

I metodi split() e join()

Il metodo join() è utile quando ci troviamo con una lista di stringhe e vogliamo unirle tra di loro per formare una nuova stringa risultante, come nel caso di una lista di parole. Questo metodo viene chiamato su una stringa che usiamo "a mo' di collante" tra le varie stringhe che vogliamo unire. Nel nostro caso questa stringa "collante" potrebbe essere una virgola seguita da uno spazio, oppure un carattere newline.

```
>>> to_do = ["portare il cane a passeggio", "finire di studiare", "fare la spesa", "lavare i panni"]
>>> ", ".join(to_do)
'portare il cane a passeggio, finire di studiare, fare la spesa, lavare i panni'
>>> da_fare = "oggi devo: " + ", ".join(to_do)
>>> da_fare
'oggi devo: portare il cane a passeggio, finire di studiare, fare la spesa, lavare i panni'

######################################################################

>>> da_fare = "\n".join(to_do)

>>> print(da_fare)
```

Col metodo split() possiamo al contrario dividere una stringa in una lista di stringhe, e dovremo passare come parametro il carattere che intendiamo usare come separatore.

```
>>> citazione = "Nel mezzo del cammin di nostra vita..."
```

```
>>> citazione.split(" ")
['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita...']
```

Il metodo format()

Per formattare le stringhe possiamo utilizzare anche il metodo format():

stringa.format(valore1, valore2, ...)

Dove stringa è la stringa da formattare e valore1, valore2, ecc. sono i valori da inserire nella stringa. Il metodo format() sostituisce ogni posizione {} nella stringa con il valore corrispondente passato come parametro.

```
>>> colore1 = "blu"
>>> colore2 = "verde"
>>> s = "I miei colori preferiti sono {} e {}.".format(colore1, colore2)
>>> print(s)


# output
I miei colori preferiti sono blu e verde.
```

Similarità tra Liste e Stringhe

I tipi di dato di Python list e string sono per certi aspetti simili tra di loro, e per questo motivo ci sono alcune tipologie di azioni che possiamo fare su entrambi. Potete in questi esempi pensare quindi alle stringhe come a delle liste di caratteri.

La funzione len()

La funzione integrata di Python len() ci permette di ottenere la lunghezza di una lista o una stringa, e quindi rispettivamente il numero di elementi o di caratteri presenti in queste:

```
>>> my_str = "spam, eggs, bacon, spam"
>>> my_list = ["spam", "spam", "spam"]

>>> len(my_str)
23

>>> len(my_list)
3
```

Operatori in e not in

Possiamo usare gli operatori in e not in per verificare se un elemento o un carattere sia presente in una lista o stringa:

```
>> my_list = ["spam", "spam", "spam"]
>>> new_str = "happiness"

>>> "bacon" in my_list
False
>>> "spam" in my_list
True
>>> "eggs" not in my_list
True
```

String Slicing

In maniera analoga a quanto visto con le liste, possiamo accedere ai caratteri di una stringa tramite indice, in modo da ottenere una porzione della stessa. In questo esempio abbiamo una variabile chiamata alfa che contiene un insieme di caratteri alfabetici + tre punti di sospensione.

Volendo possiamo isolare la porzione puramente alfabetica o i tre punti usando lo slicing e usare questi valori per creare delle nuove variabili:

```
>>> alfa = "abcdefghijklm..."


>>> dots = alfa[-3:]

>>> dots

'...'


>>> alfa = alfa[:-3]

>>> alfa

'abcdefghijklm'
```

Stringhe e cicli for

Come per le liste, possiamo usare i cicli for assieme alle nostre stringhe:

```
random_alnum = "dj1oi3u4nuqnoru01u3m3mdasd"
counter = 0
match = "d"

for char in random_alnum:
    if char == match:
        counter += 1
output = f"Ho trovato {counter} caratteri '{match}' "
>>> output
"Ho trovato 3 caratteri 'd' "
```

Tuple e Set

Le tuple in Python

Come il tipo di dato lista visto nelle lezioni precedenti, anche il tipo tuple di Python rappresenta un insieme di elementi, definiti stavolta utilizzando una coppia di parentesi tonde, oppure senza alcuna parentesi ma separati da virgola.

```
>>> tupla = (2, 4, 9, 15, 23)

>>> type(tupla)

# output

<class 'tuple'>

####################

>>> tupla

# output

(2, 4, 9, 15, 23)
```

Le tuple sono immutabili

La grande differenza tra i tipi di dato tuple e list è che le tuple sono immutabili. Per questo motivo il dato tuple non dispone di un metodo append() e non possiamo usare l'istruzione del per rimuovere elementi. Questo risulta molto comodo in certe situazioni: tenetelo a mente!

I set di Python

I set sono un altro tipo di dato di Python che permette di creare collezioni di elementi. I set vengono definiti con una coppia di parentesi graffe, e i valori sono separati da virgola

```
>>> my_set = {"asd", "qwerty", "cmd"}

>>> type(my_set)


# output

<class 'set’>
```

La caratteristica fondamentale dei set è che non possono contenere duplicati al loro interno.

Il metodo add() dei set

A differenza delle tuple possiamo però aggiungere nuovi elementi col metodo add.

```
>>> new_set = {"asd", "asd", "qwerty", "cmd"}
>>> new_set.add("cmd")
>>> new_set

# output
{'qwerty', 'cmd', 'asd'}
```

Approfondimento: Gli iterabili in Python

inora abbiamo visto diversi tipi di dato in Python, come i numeri interi, i valori booleani, le liste e le stringhe.

C’è però una caratteristica che accomuna alcuni di questi tipi di dato: le stringhe, le liste, le tuple, i set e i dizionari (che vedremo in una lezione successiva), sono tutti oggetti che possono essere trattati come sequenze che contengono elementi accessibili per essere manipolati. Questo tipo di oggetti in Python sono detti iterabili. Gli altri tipi di dato, come ad esempio i booleani e i numeri int o float non sono iterabili perché rappresentano un unico valore o una singola entità e non contengono elementi.

Funzioni per iterabili

Python fornisce diverse funzioni per manipolare gli elementi degli iterabili, vediamo alcune delle più utilizzate:

La funzione max()

La funzione max() viene utilizzata per trovare il valore massimo di un oggetto iterabile.

max(iterable, key=None, default=None)

Dove iterable è la sequenza di cui si vuole trovare il valore massimo. Ad esempio possiamo usare max() per ottenere il numero maggiore in un set di numeri:

L'argomento default specifica il valore da restituire se l'insieme di input è vuoto, mentre l’argomento key specifica un criterio di ordinamento personalizzato, ad esempio se gli passiamo len possiamo ottenere la parola più lunga in una tupla di parole:

La funzione sum()

La funzione sum() restituisce la somma degli elementi di un oggetto iterabile:

sum(iterable, start=0)

Dove iterable è l'oggetto iterabile che si desidera sommare e start è il valore di partenza opzionale della somma che per impostazione predefinita è uguale a zero, quindi se non viene specificato viene sommato il valore di tutti gli elementi. Ad esempio, sommiamo con sum() tutti gli elementi da una lista di numeri:

```
numeri = [33, 78, 5, 42, 23]
totale = sum(numeri)
```

La funzione enumerate()

La funzione enumerate() consente di iterare su una sequenza (liste, tuple, stringhe, ecc.) e restituire una tupla contenente il valore dell'elemento corrente e il suo indice:

enumerate(sequence, start=0)

Dove sequence è la sequenza da iterare e start è l'indice di partenza per la numerazione (il valore predefinito è 0). Ad esempio, se vogliamo stampare il nome di ogni frutto di una lista insieme al suo indice, possiamo utilizzare la funzione enumerate() in questo modo:

```
frutta = ["mela", "banana", "kiwi", "arancia"]

for indice, frutto in enumerate(frutta):
    print(indice, frutto)
```

La funzione sorted()

La funzione sorted() viene utilizzata per ordinare un iterabile. A differenza del metodo per liste sort(), che ha lo stesso scopo, può essere usato anche per le tuple e i set perché non modifica l'ordine degli elementi all'interno della sequenza originale ma la lascia invariata restituendo una nuova lista.

sorted(iterable, key=None, reverse=False)

Dove iterable è la sequenza di cui si vuole ordinare gli elementi. Come abbiamo visto per max(), anche in questo caso key consente di specificare un criterio di ordinamento personalizzato, mentre l'argomento reverse, se impostato su True, fa in modo che gli elementi siano in ordine decrescente.

La funzione filter()

La funzione filter() prende in input una funzione e un iterabile e restituisce un nuovo iterabile contenente solo gli elementi dell'input per i quali la funzione restituisce True.

filter(funzione, iterabile)

Vediamo un esempio in cui controlliamo se un numero è pari o non lo è:

```
def numero_pari(n):
    return n % 2 == 0
numeri = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numeri_pari = list(filter(numero_pari, numeri))


print(numeri_pari)
```

La funzione map()

La funzione map() permette di applicare una stessa funzione ad ogni elemento di un oggetto iterabile e restituisce un nuovo oggetto iterabile dello stesso tipo di quello di partenza contenente i risultati della funzione applicata a ciascun elemento.

map(function, iterable, ...)

È necessario specificare nell'argomento function la funzione che si desidera applicare a ciascun elemento dell'argomento iterable. Come indicano i puntini, la funzione può anche essere applicata a più oggetti iterabili contemporaneamente, basta passarli come argomento, e può essere utilizzata con qualsiasi funzione, anche definite dall'utente. Ad esempio, applichiamo a una lista di numeri una funzione che li triplica:

```
numeri = [1, 2, 3, 4, 5]

def triplica(n):
    return n * 3

numeri_triplicati = list(map(triplica, numeri))
print(numeri_triplicati)

# output
[3, 6, 9, 12, 15]
```

I Dizionari

I dizionari di Python

I dizionari sono simili al tipo di dato lista di cui abbiamo già parlato nelle lezioni precedenti, ma a differenza di queste, in cui ad ogni elemento corrisponde un indice progressivo quindi da 0 in su, nei dizionari ad ogni elemento è associata una chiave chiave, che potrà essere usata per accedere al valore dell'elemento.

Siamo noi a scegliere che chiave usare, e possiamo usare qualsiasi tipo di dato eccetto liste e altri dizionari. Per ogni elemento all'interno del dizionario avremo quindi una coppia chiave / valore.

I dizionari di Python vengono definiti utilizzando una coppia di parentesi graffe che contenga al suo interno delle coppie chiave / valore separate da due punti:

```
>>> mio_dict = {"mia_chiave": "mio_valore", "age": 29, 3.14: "pi greco", "primi": [2, 3, 5, 7]}
>>> type(mio_dict)
```

Dizionari e coppie chiave valore

Come vedete abbiamo utilizzato diversi tipi di valori, stringhe, interi e liste, associati ciascuno ad una chiave univoca. Le chiavi nei dizionari sono proprio il corrispettivo degli indici nelle Liste: usiamo le chiavi per richiamare i valori ad esse associati.

Per ottenere il valore associato alla chiave mia_chiave_uno facciamo in questo modo:

```
>>> mio_dict["mia_chiave"]
'mio_valore’
```

Per aggiungere un nuovo elemento al nostro dizionario facciamo in questa maniera:

```
>>> mio_dict["nuova_chiave"] = "nuovo_valore"
```

Come verificare la presenza di elementi nei dizionari

Per verificare se una chiave è presente o meno all'interno di un dizionario possiamo utilizzare gli operatori in e not in:

```
>>> "age" in mio_dict
True

>>> "zen" in mio_dict
False
```

Come rimuovere elementi da un dizionario

Per rimuovere una coppia chiave-valore possiamo utilizzare l'istruzione del:

```
>>> del mio_dict["mia_chiave"]
```

I metodi dei Dizionari di Python

Come per altri tipi di dato, anche il tipo dict dispone di alcuni metodi propri: analizziamo i principali con un nuovo esempio.

Definiamo un nuovo dizionario associato a una variabile ita_eng e inseriamoci i valori tipici di un dizionario Italiano – Inglese

ita_eng = {"ciao": "hello", "arrivederci": "goodbye", "uova": "eggs", "gatto": "cat", "arancia": "orange"}

Il metodo keys()

Keys è traducibile come "chiavi", ed è proprio il metodo che ci consente di ottenere una lista di tutte le chiavi presenti.

Quindi per ottenere una lista di tutte le chiavi presenti nel nostro dizionario ita_eng facciamo:

```
>>> ita_eng.keys()

# output
dict_keys(['ciao', 'arrivederci', 'uova', 'gatto', 'arancia'])
```

Metodo values()

Values significa valori, utilizziamo questo metodo per ottenere una lista di tutti i valori presenti:

```
>>> ita_eng.values()

# output
dict_values(['hello', 'goodbye', 'eggs', 'cat', 'orange'])
```

Il metodo items()

Infine utilizziamo il metodo items() per ottenere una lista di tutte le coppie chiavi-valore presenti.

```
>>> ita_eng.items()
```

```
# output

dict_items([('ciao', 'hello'), ('arrivederci', 'goodbye'), ('uova', 'eggs'), ('gatto', 'cat'), ('arancia', 'orange')])
```

Cosa restituiscono questi metodi?

Come possiamo interpretare dall'output, utilizzando questi metodi non ci vengono restituiti dei valori di tipi lista ma bensì, oggetti di tipo dict_keys, dict_values e dict_items.

Se abbiamo bisogno di una lista vera e propria possiamo utilizzare anche qui la nostra cara funzione list():

```
>>> parole_eng = list(ita_eng.keys())
```

```
>>> parole_eng
['ciao', 'arrivederci', 'uova', 'gatto', 'arancia']
```

Dizionari e cicli for

Possiamo inoltre utilizzare questi metodi in combinazione con un ciclo for.

Ad esempio, per mandare in print tutte le chiavi del dizionario possiamo fare:

```
for chiave in ita_eng.keys():
    print(chiave)


# output
ciao
arrivederci
uova
gatto
arancia
```

Il metodo get()

Ma cosa succederebbe qualora provassimo a richiamare un valore associato a una chiave che non esiste nel nostro dizionario? Come è facile ipotizzare otterremmo un errore, e nello specifico un'eccezione del tipo KeyError, che se non gestita causerà il crash del programma.

```
>>> ita_eng["birra"]


# output
Traceback (most recent call last):
  File "stdin", line 1, in module
KeyError: 'birra’
>>> ita_eng.get("birra", "Chiave non trovata, mi spiace!")
```

Il Metodo setdefault()

Parallelamente a quanto fatto col metodo get(), ci saranno dei momenti in cui a seconda della complessità di ciò che stiamo facendo, potremmo avere la necessità di creare una coppia chiave valore qualora una chiave non sia già presente e associata a un valore nel dizionario.

Restando sull'esempio precedente, supponiamo di voler verificare se sia presente la chiave birra nel nostro dizionario e di volerla aggiungere assieme ad un valore di default qualora non sia già presente

```
>>> ita_eng
{'ciao': 'hello', 'arrivederci': 'goodbye', 'uova': 'eggs', 'gatto': 'cat', 'arancia': 'orange'}

>>> ita_eng.setdefault("birra", "beer")
"beer"

>>> ita_eng
{'ciao': 'hello', 'arrivederci': 'goodbye', 'uova': 'eggs', 'gatto': 'cat', 'arancia': 'orange', 'birra':
'beer'}
```

Gestione degli Errori

Eccezioni di Python e Messaggi di Errore

Parlando di errori mi riferisco in realtà a quelle eccezioni (in inglese exception) di Python che quando si manifestano, comportano spesso il crash del programma e la comparsa di un messaggio di errore. Otteniamo ad esempio un'eccezione del tipo NameError, provando a moltiplicare una variabile che non è stata definita.

```
>>> z * 5

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

Un altra tipologia di errore ci viene mostrata se proviamo ad effettuare una divisione per zero, che restituisce ZeroDivisionError:

```
>>> 3/0

ZeroDivisionError: division by zero
```

Questi messaggi di errore sono molto comodi per noi sviluppatori, in quanto fornendoci dettagli relativi a un comportamento anomalo del nostro programma, ci permettono di risolvere i vari bug che inevitabilmente si presentano.

Ma non tutti gli errori sono bug.

Ad esempio è più che normale ottenere ZeroDivisionError provando a fare 3 / 0.

In casi come questi è quindi importante gestire queste eccezioni, per rendere i nostri programmi più robusti e user friendly.

Le istruzioni try ed except in Python

Le istruzioni principali che vengono usate per questo scopo sono try ed except, che possono essere tradotte rispettivamente come prova e eccetto o a accezione di.

Volendo potete pensare a queste come a degli if / else ideati per la gestione delle eccezioni, in quanto anche in questo caso definiremo dei blocchi di codice distinti.

Il codice del blocco try verrà eseguito qualora tutto andasse liscio e senza errori, mentre il codice inserito nel blocco except verrà eseguito solamente qualora si dovesse manifestare l'eccezione ad esso associato.

Abbiamo qui di seguito una funzione moltiplicatore che richiede all'utente di inserire dei valori per le variabili a e b, ed effettua quindi una moltiplicazione.

Proviamo ad eseguire il codice e a forzarne il comportamento ipotizzato introducendo dei caratteri non numerici all'interno delle variabili a e b:

```
def moltiplicatore():
    a = int(input('Inserisci il valore di a: '))
    b = int(input('Inserisci il valore di b: '))
    risultato = a * b
    print(risultato)



moltiplicatore()


# output
Inserisci il valore di a: eggs
ValueError: invalid literal for int() with base 10: 'eggs'
```

```
def moltiplicatore():
    try:
        a = int(input("Inserisci il valore di a: "))
        b = int(input("Inserisci il valore di b: "))
        risultato = a * b
        print(risultato)
    except ValueError:
        print("'Hey tu! solo caratteri numerici, grazie!")
moltiplicatore()
# output
Inserisci il valore di a: bacon
Hey amico! solo caratteri numerici, grazie!
```

L'istruzione finally

Un'altra istruzione di Python utile in questi contesti è l'istruzione finally, traducibile in italiano come alla fine o infine. Come il nome stesso suggerisce, il codice definito nel blocco di finally verrà eseguito alla fine del programma qualsiasi cosa succeda, sia che si manifesti un errore oppure no.

Modifichiamo la nostra funzione usando questa istruzione:

```
def moltiplicatore():
    try:
        a = int(input("Inserisci il valore di a: "))
        b = int(input("Inserisci il valore di b: "))
        risultato = a * b
        print(risultato)
    except ValueError:
        print("Hey tu! solo caratteri numerici, grazie!")
    finally:
        print("Sto chiudendo l'applicazione!")

moltiplicatore()
```

Come mostrare i messaggi di errore

Un'altra cosa che possiamo fare per migliorare il codice è mostrare in output il messaggio di errore.

Nel blocco dell'except, dopo il nome dell'eccezione che stiamo gestendo, possiamo aggiungere as e, e quindi mandare in output il valore di e tramite print():

```
def moltiplicatore():
    try:
        a = int(input("Inserisci il valore di a: "))
        b = int(input("Inserisci il valore di b: "))
        risultato = a * b
        print(risultato)
    except ValueError as e:
        print(f"Errore: {e}")
        print("Hey tu! solo caratteri numerici, grazie!")
    finally:
        print("Sto chiudendo l'applicazione!")


moltiplicatore()
```

Ambienti Virtuali in Python con venv

Il Python Package Index

Il Python Package Index è un deposito di software scritto per Python.

Si tratta di una risorsa molto importante perché contiene al suo interno centinaia di migliaia di progetti scritti da varie organizzazioni o singoli developer, e non parlo solo di semplici progetti sviluppati in un fine settimana ma di librerie complesse e strutturate utilizzate da migliaia di aziende e altre attività, nel mondo reale. Da questo repository è possibile scaricare queste librerie gratuitamente e iniziare a creare software anche molto avanzato, senza dover reinventare la ruota.

Il comando pip list

I moduli del Python Package Index vengono installati da terminale, tramite un programma chiamato PIP, che dovreste già avere nel vostro computer in quanto è presente, tipicamente, nel pacchetto di installazione di Python. I vari moduli possono essere installati in due aree: a livello di sistema o in un ambiente virtuale.

Partiamo dall'analisi di quanto presente a livello di sistema: aprite quindi un terminale e date il comando pip list.

Cosa sono gli Ambienti Virtuali di Python?

Gli Ambienti Virtuali o Virtual Environments di Python sono uno strumento che permette di creare degli spazi indipendenti dal resto del sistema in cui è possibile testare e lavorare con Python e pip.

Concretamente avremo una cartella che conterrà vari file necessari al funzionamento dell'ambiente e una copia dei binary di Python, e dentro questa potremo poi installare tutti i moduli con la versione che vogliamo in base al progetto su cui stiamo lavorando, ma che richiederanno l'attivazione di questo ambiente per essere utilizzati.

Esistono vari strumenti per creare ambienti virtuali, e noi useremo uno chiamato venv che è presente di default con l'installazione di Python.

Creare Ambienti Virtuali Python su Windows, Linux e Mac OS

È importante sapere quale comando utilizzare per poter richiamare una shell di Python da terminale. Dovrete in sostanza usare lo stesso comando usato finora nel corso, con l'aggiunta di alcune informazioni aggiuntive che informino Python che volete creare un ambiente virtuale. Spostatevi quindi in una cartella dove avete permessi di scrittura (per testare va benissimo anche la cartella Documenti o Scrivania/Desktop) e date il comando:

```
python -m venv venv
```

Così facendo abbiamo detto a Python (comando: python o python3) che intendiamo usare il modulo venv (-m venv) per creare un ambiente virtuale in una nuova cartella chiamata venv. Avremmo potuto quindi cambiare la parte finale del comando per creare un ambiente in una cartella chiamata in qualsiasi modo.

Creare Ambienti Virtuali con Conda

Un altro sistema di gestione di pacchetti e di ambienti virtuali molto utilizzato è Conda, che rispetto a venv offre diverse funzionalità aggiuntive, come la possibilità di creare ambienti virtuali con diverse versioni di Python o una gestione delle dipendenze avanzata. Per creare un nuovo ambiente virtuale tramite Conda, su Anaconda/Miniconda Prompt digitate:

```
conda create --name myenv
```

Sostituite myenv con un nome a vostra scelta. Se necessario potete indicare quale versione di Python specifica installare alla fine del comando:

```
conda create --name myenv python=3.11
```