



# **Andrea Bertolini**

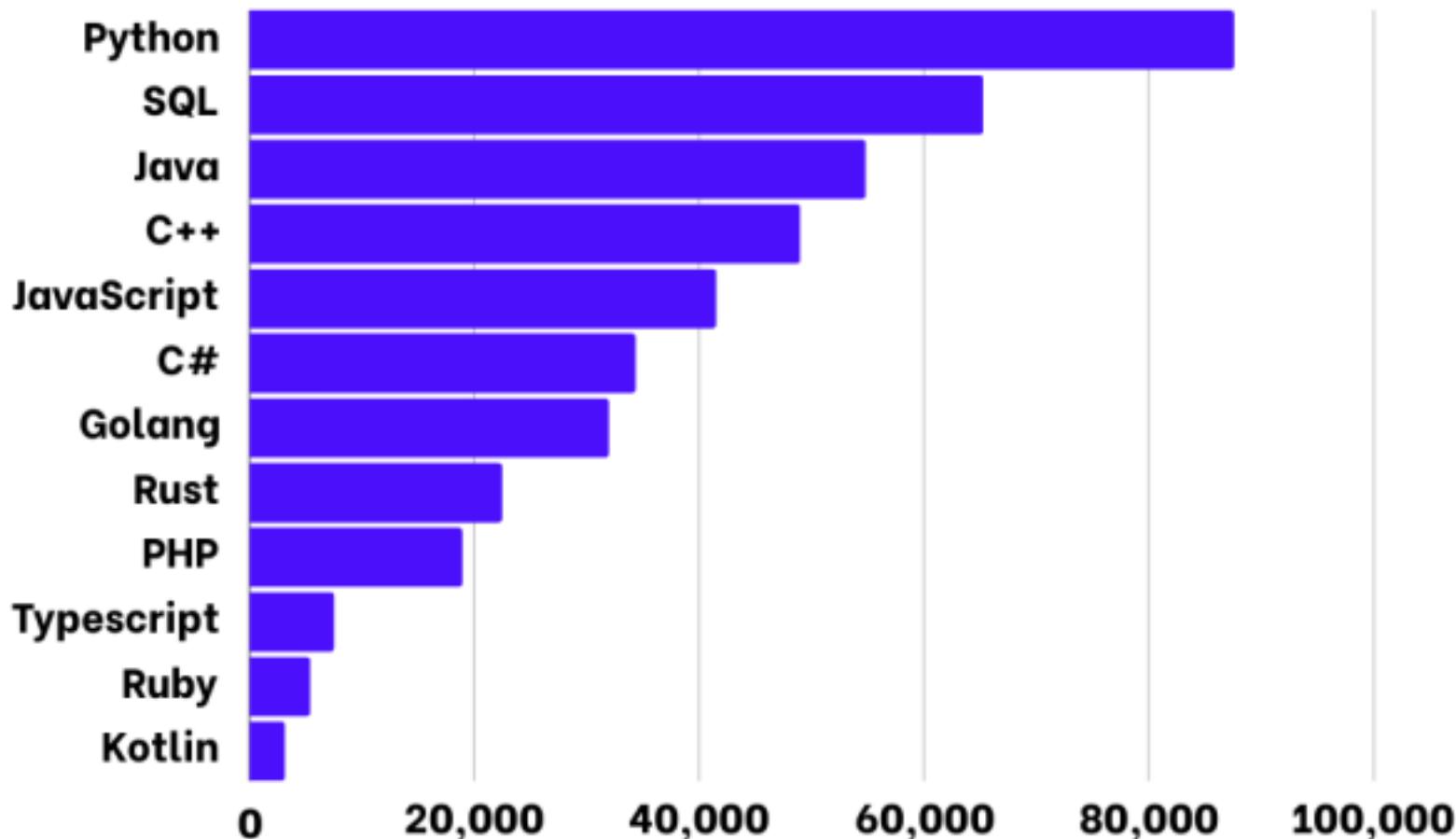
CLOUD COMPUTING - Sviluppo applicazioni web  
lato server in Python con metodologia devOps

## **Sottomodulo 6.1**



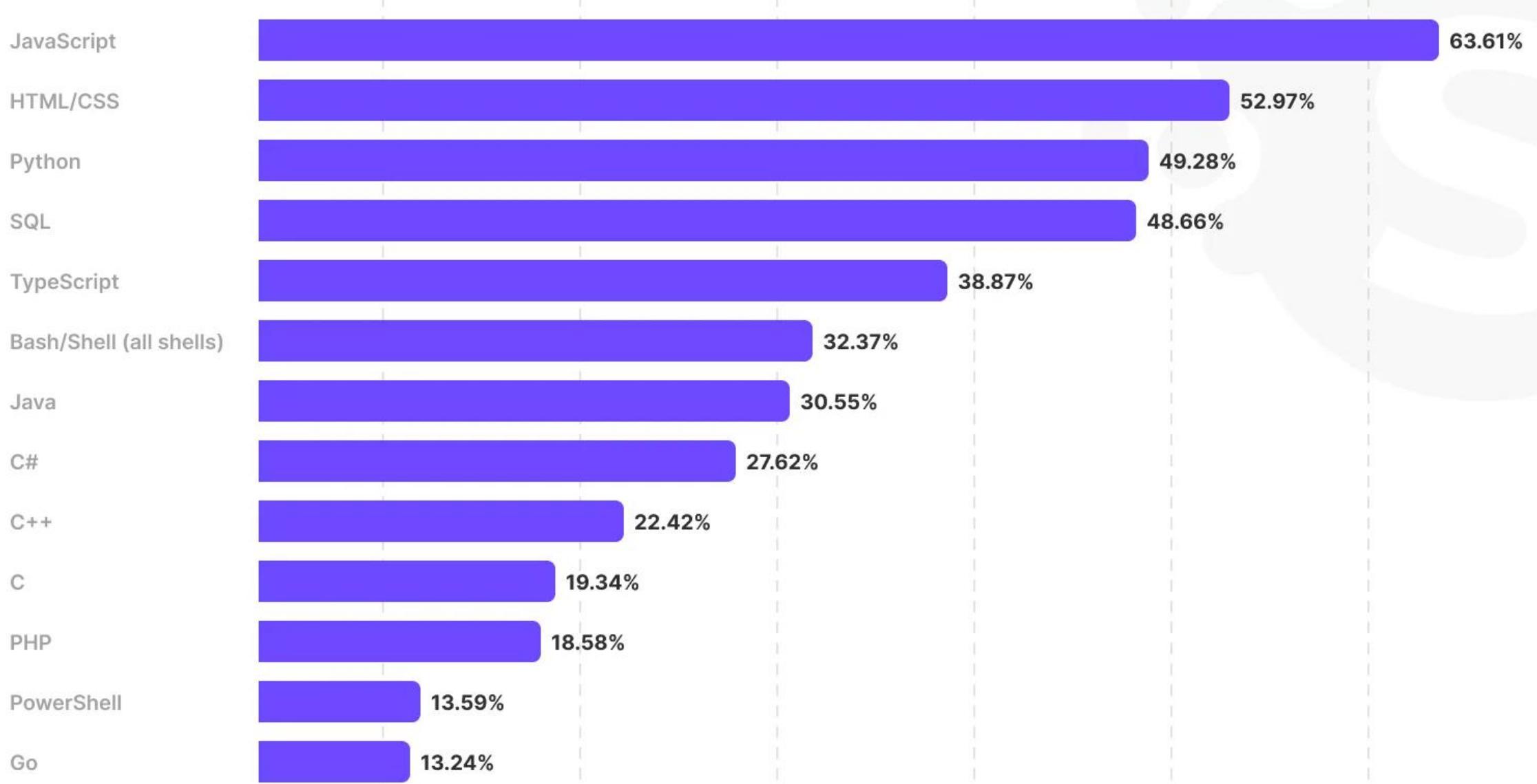
# 2024 Programming Language Breakdown

# of open job postings for each language, in the US only





# Top programming languages and frameworks for 2024





# Why Python?



**Web Development**



**Computer Graphics**



**Testing**



**Web Scraping**



**Data Analysis**



**Machine Learning**



**Big Data**



**Internet of Things**





# Introduzione

Il linguaggio Python in generale.

Overview.

Getting Started.



# Il linguaggio Python in breve

È un linguaggio di programmazione **general-purpose**, di **alto livello** ed è utilizzato ampiamente in tantissimi settori.

Con il termine **general-purpose** si intende che Python può essere utilizzato per creare applicazioni di Data Science, Machine Learning, Scripts, Web, Desktop etc.

Con il termine **alto livello** si intende che il linguaggio è facilmente comprensibile da un essere umano.



# Introduzione al Python

## **Procedurale**

si specifica la procedura da eseguire sui dati

## **Strutturato**

concetto di visibilità delle variabili

## **Dynamically-Typed**

Il tipo del dato viene dedotto da Python sulla base dei valori forniti.

## **Interpretato**

il codice scritto viene eseguito da un interprete e non direttamente dalla CPU.



# Perché Python

- Essendo un linguaggio interpretato, Python è facile da imparare ed utilizzare, grazie anche all'interazione dinamica con l'interprete.
- Supportati i vari paradigmi di programmazione, come ad esempio quello **funzionale** e **orientato agli oggetti**.
- Viene posta particolare attenzione alla leggibilità del codice, alla facilità e rapidità di scrittura.
- I programmatorei possono trarre le loro idee in software con più facilità utilizzando meno righe di codice.
- Ampia disponibilità di librerie software già pronte all'uso.



# Perché Python

## 1. Semplicità

Python è uno dei linguaggi più semplici con cui avvicinarsi al mondo della programmazione. **La sua semplicità non limita affatto le sue potenzialità!**

- Python è un linguaggio gratuito e open-source sviluppato mediante C
- Facile da imparare
- Facile da leggere
- Interpretato
- Dynamically-Typed



# Perché Python

## 1. Semplicità

Python è un linguaggio interpretato, o meglio..

Ha un compilatore interno che interpreta ogni riga che voi scrivete e la trasforma in byte-code, un linguaggio intermedio che viene poi eseguito da un'unità software chiamata **Python Virtual Machine**.

Agli occhi dell'utente non vi è alcun processo di compilazione: il codice è scritto ed eseguito. Per dettagli vedere la Python Developer Guide.



# Perché Python

## 1. Semplicità

Python è un linguaggio **dynamically-typed**:

ciò significa che non è necessario esplicitare il tipo di dato di una variabile in fase di dichiarazione come avviene invece per altri linguaggi quali Java, C, C++ ..

Il tipo di dato viene dedotto automaticamente da Python a run time e può mutare con l'esecuzione del programma. **Secondo voi questa tecnica ha degli svantaggi?**



# Perché Python

## 1. Semplicità

Java

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world");  
    }  
}
```

Python

```
print("Hello, world")
```

It's that **SIMPLE!**



# Perchè Python

## interpretato

- Non compilato come Java, C, C++
- Codice scritto ed eseguito direttamente da un **interprete**
- E' possibile scrivere comandi direttamente nell'interprete e osservarne il risultato (come in R)

Java:



Python:





# Perché Python

## 1. Librerie e Framework

A causa della sua popolarità, Python ha centinaia di librerie e framework che aiutano a velocizzare il processo di sviluppo del programma.

Grande community di sviluppatori in attività.

È il linguaggio di programmazione di punta nel campo dell'intelligenza artificiale.



# Alcuni esempi..

1. Numpy, Scipy, Pandas (ambito scientifico)
2. Django, Flask (sviluppo web)
3. Plotly, Seaborn, Matplotlib (diagrammi, schemi)
4. Tensorflow, Pytorch, Keras (intelligenza artificiale)
5. Kivy (sviluppo mobile)
6. Tkinter, pyQt5 (interfacce grafiche)



# Applicazioni reali che utilizzano Python

1. Youtube
2. Instagram
3. Spotify
4. Dropbox
5. Netflix
6. Reddit
7. TikTok
8. Pinterest



# I falsi miti riguardanti Python

## ***Myth #1: Python Is Merely a Scripting Language***

Vi è molta confusione sul termine scripting language. Python è un linguaggio di programmazione completo, sempre in evoluzione, con cui si può implementare qualsiasi tipo di software.

## ***Myth #2: Python Is Slow***

È facile cedere alla tentazione di affermare che, essendo Python interpretato, questo sia più lento di linguaggi compilati quali C o C++. La verità, però, è che la velocità di esecuzione dipende da tanti fattori: implementazione di Python, scrittura del programma, scelta delle librerie sono tutti aspetti che hanno un impatto diretto sulle performance.



# I falsi miti riguardanti Python

## ***Myth #3: Python Cannot Be Compiled***

C'è una sorprendente quantità di dibattiti aperti sul termine **compiled**, in gergo tecnico è meglio chiamare i linguaggi come C o C++ linguaggi **assemblati** (assembled). Ci sono tecniche per compilare Python!

## ***Myth #4: Python Is Unsuitable for Large Projects***

La gestione di grandi progetti è semplificata rispetto ai linguaggi come C o C++. Un software scritto in Python è strutturabile in pacchetti e moduli importabili a piacere che, se usati correttamente, permettono di creare codebases ben strutturate ed organizzate.



# I falsi miti riguardanti Python

## ***Myth #5: Python Is So Simple That Is NOT a Programming Language***

Il fatto che Python sia così user-friendly viene spesso denigrato da alcuni programmatori abituati ad utilizzare linguaggi più di basso livello. Lo stesso si potrebbe dire di loro se comparassimo la programmazione moderna con la programmazione in assembly.



# Installing Python

## Windows:

- Download Python from <http://www.python.org>
- Install Python.
- Run **Idle** from the Start Menu.

## Ambienti preconfigurati:

- **Google Colab** è un ambiente web che permette di utilizzare Python su browser senza installare nulla.

## Mac OS X:

- Python is already installed.
- Open a terminal and run `python` or run Idle from Finder.

## Linux:

- Chances are you already have Python installed. To check, run `python` from the terminal.
- If not, install from your distribution's package system.



# Installing Python

## Active Python Releases

For more information visit the [Python Developer's Guide](#).

Python version	Maintenance status	First released	End of support
3.13	prerelease	2024-10-01 (planned)	2029-10
3.12	bugfix	2023-10-02	2028-10
3.11	bugfix	2022-10-24	2027-10
3.10	security	2021-10-04	2026-10
3.9	security	2020-10-05	2025-10
3.8	security	2019-10-14	2024-10

Nel prompt dei comandi digitate: `py --version` per vedere la vostra versione installata.



# Getting started

## Finding an Interpreter

Before we start Python programming, we need to have an interpreter to interpret and run our programs.

- There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) that comes bundled with the Python software downloaded from <http://python.org>.  
Examples: Spyder, Pycharm, Jupyter Notebook, etc.
- Online interpreters like <https://ide.geeksforgeeks.org> that can be used to run Python programs without installing an interpreter.
- Anaconda (<https://www.anaconda.com>) – a distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment.



# Getting Started

- Utilizzeremo Visual Studio Code e Google Colab



- Un altro IDE diffuso è PyCharm, la cui versione gratuita è chiamata community edition.





# The Python Interpreter

- Allows you to type commands one-at-a-time and see results
- A great way to explore Python's syntax
  - Repeat previous command: Alt+P

A screenshot of the Python Shell window. The title bar reads "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window displays the Python version information: "Python 2.4.3 (#69, Mar 29 2006, 17:35:34) [MSC v.1310 32 bit (Intel)] on win32" and a copyright notice. It also shows a warning about a personal firewall. At the bottom, there is a code editor window for IDLE 1.1.3, containing the following Python code:

```
>>> print "Hello there"
Hello there
>>> print "How are you"
How are you
>>> |
```

The status bar at the bottom right indicates "Ln: 16 Col: 4".



# Our First Python Program

- Python **non ha** un metodo `main` come Java
- Ogni riga Python **non necessita del punto e virgola!**

**hello.py**

```
1 print("Hello, world!")
```



# The print Statement

```
print ("text")
print()          (a blank line)
```

- Escape sequences come \" sono le stesse di Java
- Le Strings possono anche cominciare/terminare con '

## swallows.py

```
1 print("Hello, world!")
2 print()
3 print("Suppose two swallows \"carry\" it together.")
4 print('African or "European" swallows?')
```



# Scrittura di programmi su file

- La programmazione vera e propria viene fatta creando dei file di comandi (*programmi* o *script*)
- Un file di comandi Python può essere creato con qualsiasi editor di testi
- Per poter essere importati come moduli (vedremo), i file Python devono avere l'estensione `.py` (attenzione agli editor che aggiungono estensioni automatiche ai file salvati)
- Un file di comandi può essere eseguito in modalità *batch* tramite l'interprete Python
- L'interprete esegue i comandi contenuti nel file uno dopo l'altro e termina.
- A differenza dell'uso interattivo, in questo caso l'interprete non stampa automaticamente l'output dei comandi (usare esplicitamente `print` a questo scopo)



# Oggetti

## Ogni cosa è un oggetto

- Python manipola oggetti
- Ad ogni oggetto è associato un *tipo*.
- Python fornisce una serie di tipi di oggetti predefiniti. I principali sono:

numeri 6.345

stringhe "The meaning of life"

liste ["a", "b", 23]

dizionari { "UNO" : 1, "DUE" : 2}

tuple ("basso", "medio", "alto")

file myfile = fopen("file.txt", "r")

None indica l'oggetto "nullo"

- Gli oggetti possono essere manipolati tramite operatori (5.3 + 4.2) e funzioni (print("print me"))



# Oggetti

## Creazione di oggetti

- Un oggetto si crea “assegnando” un valore ad una *variabile* che rappresenterà un *riferimento* all’oggetto stesso:

```
>>> a = 12 * 3  
>>> b = "sono una stringa"
```

- Il tipo dell’oggetto viene stabilito al momento della creazione, e dipende dal valore che gli viene assegnato:

```
>>> a = 12 * 3 # number  
>>> b = "sono una stringa" # string
```



# Oggetti

## Operazioni su oggetti

- Le operazioni che possono essere fatte (ed il loro risultato) dipendono dal tipo degli oggetti coinvolti:

```
>>> a = 4
>>> b = a * 3                      # number
>>> print(b)
12
>>> c = "sono"
>>> d = " una stringa"
>>> e = c + d                      # string
>>> print(e)
sono una stringa
```



# Variabili

## Identifieri di oggetti

- Una variabile rappresenta un riferimento ad un oggetto in un programma Python
- Una variabile è una sequenza arbitraria di:
  - lettere maiuscole ([A..Z])
  - lettere minuscole ([a..z])
  - cifre ([0..9])
  - underscore (-)

in cui il primo carattere non sia una cifra

## Esempi

- Corrette:

a b1 \_c4 A45b var\_

- Scritte:

```
12a      # comincia per cifra  
var$      # carattere non ammesso
```



# Variabili

## Scelta del nome

- E' utile dare alle variabili un nome che ricordi la loro funzione (`year = 2008`)
- Nomi troppo lunghi possono essere scomodi se la variabile deve essere scritta spesso nel programma

## Keywords

Python ha una serie di keyword *riservate* che non possono essere usate come identificatori (vedremo):

```
and      del      from     not      while
as       elif     global    or       with
assert   else     if        pass     yield
break   except   import   print
class    exec    in        raise
continue finally  is        return
def     for     lambda   try
```



# Variabili: tipizzazione dinamica

- Python utilizza una forma di tipizzazione *dinamica*:
  - Solo gli oggetti hanno tipi
  - il tipo di una variabile è il tipo dell'oggetto a cui si riferisce
- se ad una variabile si assegna un nuovo oggetto, il suo tipo diventa quello del nuovo oggetto a cui si riferisce (mentre il vecchio oggetto rimane invariato)

```
>>> a = 4                      # number
>>> a = "stringa"      # string
>>> b = a
>>> print(b)
stringa
>>> b = 3
>>> print(b)
3
>>> print(a)
stringa
```



# Variabili: tipizzazione dinamica

- In ogni istante, una variabile ha il tipo dell'oggetto a cui si riferisce
- Non è possibile eseguire operazioni sulla variabile che non siano compatibili con il suo tipo (non c'è conversione implicita di tipo)

```
>>> a = 4                      # number
>>> b = "stringa"      # string
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
 'int' and 'str'
```



# Commenti

## Descrizione

- Un commento comincia con il carattere '#'
- Tutto ciò che segue fino al termine della riga viene ignorato dall'interprete:

```
>>> a = "aa"  
>>> a = a * 2      # replica due volte a  
>>> print(a)      # stampa a  
aaaa
```

- I commenti servono a favorire la comprensione del codice dopo averlo scritto, o da parte di altri.



# Commenti

- Syntax:

**# comment text (one line)**

**swallows2.py**

```
1 # Suzy Student, CSE 142, Fall 2097
2 # This program prints important messages.
3 print("Hello, world!")
4 print()                      # blank line
5 print("Suppose two swallows \"carry\" it together.")
6 print('African or "European" swallows?')
```



# Commenti

- Syntax:

```
"""
```

**This is a multiline comment.**

```
"""
```

## swallows2.py

```
1  """ Suzy Student, CSE 142, Fall 2097
2      This program prints important messages.
3  """
4  print("Hello, world!\n")
5  print("Suppose two swallows \"carry\" it together.")
6  print('African or "European" swallows?')
```



# Commenti

I commenti sono ignorati da Python, ma sono utili al programmatore per spiegare il codice sorgente qualora fosse necessario.

- Attenti a non abusarne! Preferite un codice chiaro e leggibile, con il nome delle variabili che ricordano ciò che contengono, ad un codice tutto commentato.
- Utilizzate i commenti anche per disabilitare parti di codice che servono solo da debug.



# Introduzione ai tipi di dato

Tipi di dato built-in numerici e stringhe



# Tipi di dato numerici

**interi normali** interi con una gamma prefissa di possibili valori (che dipende dall'architettura, e.g. [-2147483648,2147483647] con parole a 32-bit):

3452, -15, 0

**interi lunghi** interi con gamma illimitata (dipende dalla quantità di memoria della macchina):

3422252352462462362446,  
-134135545444432435

**reali in virgola mobile** numeri reali, rappresentabili sia in notazione scientifica che non:

-1.234, 19e5, -2E-6, 45.4E+24

**booleani** assumono solo due valori: `True` (corrispondente a 1) e `False` (corrispondente a 0). Si usano nelle espressioni logiche.



# Operazioni su numeri

operatori aritmetici binari: + - \* /

- l'interprete esegue operazioni solo su operandi dello stesso tipo.
- Quando gli operandi sono di tipo diverso, l'interprete li converte nel tipo più complesso tra i tipi degli operandi prima di eseguire l'operazione

```
>>> 3 / 2      # operandi interi -> divisione intera  
1  
>>> 3 + 4.5   # primo operando convertito in reale  
7.5  
>>> 3.2 / 2   # secondo operando convertito in reale  
1.6  
>>> 3 / 1.2   # primo operando convertito in reale  
2.5
```



# Operazioni su numeri

## divisione intera: //

- la *divisione intera* restituisce il quoziente della divisione intera a prescindere dal tipo degli operandi numerici
- se di diverso tipo, gli operandi vengono comunque prima convertiti nel tipo più complesso

```
>>> 3 // 2  
1  
>>> 3.2 // 2  
1.0
```



# Operazioni su numeri

## modulo: %

- l'operazione di *modulo* restituisce il resto della divisione intera
- funziona sia per operandi interi che reali, nel secondo caso il resto sarà un numero reale

```
>>> 3 % 2  
1  
>>> 4.5 % 2  
0.5
```



# Operazioni su numeri

## elevamento a potenza: \*\*

- eleva il primo operando alla potenza del secondo
- effettua automaticamente la conversione di tipo del risultato se necessario

```
>>> 4 ** 2  
16  
>>> 4 ** 2.5  
32.0  
>>> 4 ** -1  
0.25
```



# Operazioni su numeri

operatori di confronto: < <= > >= == !=

- Confrontano due operandi e restituiscono un valore di verità

```
>>> a = 3
>>> b = 4
>>> a <= b      # a minore o uguale a b
True
>>> -b > a      # -b maggiore di a
False
>>> a != b      # a diverso da b
True
```



# Operazioni su numeri

## Uguaglianza vs assegnazione

L'operatore di uguaglianza (==) non va confuso con l'istruzione di assegnazione (=)

```
>>> a = 3
>>> b = 4
>>> a == b      # a uguale a b
False
>>> a = b      # a prende il valore di b
>>> print(a)
4
```



# Operazioni su numeri

operatori logici: and or not

- Permettono di costruire espressioni logiche di cui testare il valore di verità:

```
>>> a == b and a != b
```

```
False
```

```
>>> a == b or not a == b
```

```
True
```

Cosa è vero e cosa è falso

**Falso** False, None, lo zero (di qualunque tipo), un oggetto vuoto (stringa, tupla, etc)

**Vero** tutto il resto



# Operazioni sui numeri

- Quando si scrivono espressioni contenenti combinazioni di operatori, l'ordine con cui le operazioni sono eseguite dipende dalla *precedenza* degli operatori
- gli operatori ordinati per precedenza decrescente sono:
  - \* % / //
  - + -
  - < <= > >= == <> !=
  - not
  - and
  - or
- quando due operatori hanno stessa precedenza, l'ordine è da sinistra a destra
- è sempre possibile (e più facile) forzare l'ordine delle operazioni raggruppando tra parentesi tonde le sottoespressioni



# Operazioni sui numeri

- Una variabile viene creata nel momento in cui le si assegna un valore.

```
>>> a = 4 * 2 / 3
```

- Non è possibile utilizzare una variabile prima che venga creata

```
>>> print(b)
```

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

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

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

- Una successiva istruzione di assegnazione modificherà il contenuto di una variabile con il risultato dell'espressione a destra dell'assegnazione

```
>>> a = 4 * 5
```

```
>>> b = 2
```

```
>>> a = a ** b
```

```
>>> print(a)
```

```
400
```



# Stringhe: rappresentazione di testi

## Apici singoli e doppi

- Una stringa può essere delimitata da apici *singoli* o *doppi*

```
>>> "abc"  
'abc'  
>>> ' abc'  
' abc'
```

- l'eco dell'interprete (salvo casi particolari) restituisce una stringa delimitandola con apici singoli
- I due tipi di rappresentazione sono utili per scrivere stringhe contenenti l'altro tipo di apice:

```
>>> "l' alba"  
"l' alba"  
>>> 'tra "apici"'  
'tra "apici'"
```



# Stringhe: rappresentazione

## Sequenze di *escape*

- Le stringhe possono contenere alcuni caratteri speciali di controllo.
- Questi caratteri vengono preceduti dal carattere \ (*backslash*) generando una *sequenza di escape*.
- Sequenze di escape si usano anche per poter scrivere caratteri come apici, e la '\' stessa.
- Le più comuni sequenze di escape sono:

\\" backslash (scrive \ )  
\' apice singolo (scrive ' )  
\\" apice doppio (scrive " )  
\n ritorno a capo  
\t tab orizzontale



# Stringhe: rappresentazione

## Stringhe *raw*

- In alcuni casi (e.g. percorsi di file in sistemi Windows), la presenza delle sequenze di escape complica le cose
- E' possibile specificare che una stringa deve essere letta così com'è aggiungendo il prefisso `r` (per *raw*):

```
>>> print("C:\\system\\tmp")
C:\\system mp
>>> print(r"C:\\system\\tmp")
C:\\system\\tmp
```



# Stringhe: rappresentazione

- Per poter rappresentare una stringa che abbia più righe con le modalità precedenti, è necessario scriverla su un'unica riga inserendo sequenze di escape \n.
- E' possibile invece scrivere stringhe su più righe, delimitandole con virgolette triple (usando sia singoli che doppi apici)

```
>>> multiriga = """vai a capo con invio
... l'interprete chiede altro testo
... finisci con tre doppi apici """
>>>
>>> print(multiriga)
vai a capo con invio
l'interprete chiede altro testo
finisci con tre doppi apici
```

- nelle sessioni interattive, se l'interprete si aspetta del testo su più righe, presenta ... invece di >>>.



# Stringhe: rappresentazione

## Operatori

Somma tra stringhe concatena due stringhe e restituisce il risultato

```
>>> "biologia" + "molecolare"  
'biologiamolecolare'
```

Prodotto stringa per intero replica la stringa un numero di volte pari all'intero, e restituisce la stringa risultante (e.g. utile per stampare delimitatori)

```
>>> "=" * 20  
'=========='
```



# Stringhe come sequenze

## Operazioni su sequenze

- Una stringa è una **sequenza** di caratteri
- Una sequenza è una collezione di oggetti (in questo caso caratteri) con ordinamento posizionale (da sinistra a destra)
- Questa caratteristica fa sì che le stringhe supportino una serie di operazioni su sequenza
- Ad esempio si può calcolare la lunghezza di una stringa (numero di caratteri):

```
>>> len("abcd")  
4  
>>> len("abcd\n ") # \n vale 1 carattere  
6
```



# Operazioni su sequenze

## Ricerca

- L'operatore di confronto `in` restituisce `True` se un certo carattere (o sottostringa) si trova in una stringa, `False` altrimenti

```
>>> s = "abcd"  
>>> "a" in s  
True  
>>> "bc" in s  
True  
>>> "bcd" in s  
True  
>>> "ad" in s  
False
```



# Operazioni su sequenze

## Indicizzazione

- E' possibile recuperare un carattere all'interno di una stringa specificandone la posizione (a partire da zero):

```
>>> s = "abcd"  
>>> s[0]  
'a'  
>>> s[3]  
'd'  
>>> s[4]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range  
• Superare gli estremi della stringa genera un errore
```



# Operazioni su sequenze

- E' possibile specificare posizioni negative, che si contano da destra a sinistra (a partire da -1):

```
>>> s = "abcd"  
>>> s[-1]  
'd'  
>>> s[-3]  
'b'  
>>> s[-4]  
'a'
```

- La posizione può essere specificata tramite un'espressione che restituisca un valore numerico

```
>>> a = -3  
>>> b = 1  
>>> s[a+b]  
'c'
```



# Operazioni su sequenze

## Sottostringa

- Permette di restituire porzioni di stringa (*slices*)
- $x[i:j]$  restituisce la sottostringa che va dalla posizione  $i$  *inclusa* alla  $j$  *esclusa*

```
>>> s = "abcd"  
>>> s[1:3]  
'bc'
```

- $i$  vale 0 se non specificato.  $j$  vale la lunghezza della stringa se non specificato.

>>> s[1:len(s)]	>>> s[:3]
'bcd'	'abc'
>>> s[1:]	>>> s[:-1]
'bcd'	'abc'
>>> s[0:3]	>>> s[:] # copia stringa
'abc'	'abcd'



# Operazioni su sequenze

## Operazioni su altri tipi di sequenze

- Le operazioni su sequenze sono definite su tutti i tipi sequenza
- Anche liste e tuple sono sequenze (vedremo).

```
>>> a = [1, 2, 3, 4]
>>> len(a)
4
>>> a[-2]
3
>>> a[:-2]
[1, 2]
>>> a * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> a + [5, 6]
[1, 2, 3, 4, 5, 6]
```



# Immutabilità delle stringhe

## Oggetti immutabili

- Le stringhe sono un tipo di oggetti **immutabili**
- Non è possibile modificare una stringa una volta creata
- Le operazioni su stringhe restituiscono sempre un *nuovo* oggetto con il risultato. Il *vecchio* oggetto rimane invariato

```
>>> s = "abcd"  
>>> s * 2  
'abcdabcd'  
>>> s  
'abcd'
```



# Immutabilità delle stringhe

- Non è possibile quindi sostituire un carattere all'interno di una stringa

```
>>> s[2] = "h"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- E' sempre possibile però assegnare la nuova stringa creata alla variabile che si riferiva alla vecchia stringa

```
>>> s = "abcd"
>>> s = s * 2
>>> s
'abcdabcd'
```

- Non stiamo modificando la vecchia stringa, ma solo cambiando l'oggetto a cui la variabile si riferisce



# Conversioni di tipo

## Operazioni tra stringhe e numeri

- Abbiamo detto che non è possibile eseguire operazioni tra tipi non compatibili (e.g. sommare una stringa e un numero)
- E' possibile però convertire esplicitamente un oggetto di un tipo in uno di un altro tipo (dove tale conversione è definita)

```
>>> "x=" + str(12)      # str da numeri a stringhe  
'x=12'  
>>> int("3") + 4       # int da stringhe a interi  
7  
>>> float("3.2") / 1.2 # float da stringhe a reali  
2.666666666666667
```



# Formattazione di stringhe

## Operatore %

- L'operatore % serve a formattare una stringa a partire da una tupla di argomenti.

```
>>> "nome=%s, lunghezza=%d" % ("7rsa", 356)
'nome=7rsa, lunghezza=356'
```

- La stringa a sinistra contiene degli specificatori di conversione (%s), la tupla a destra i valori da sostituirci
- %s indica una stringa, in pratica utilizza la funzione `str` per convertire l'elemento corrispondente della tupla in una stringa

Ma non è finita qui! Vediamo direttamente sul codice altri modi più efficienti di formattare stringhe..



# Variabili Booleane

Tutte le volte che dovete interrogare variabili, salvare l'esito di una operazione etc.. dovete utilizzare variabili booleane, il cui contenuto può essere solamente True o False.

```
max_time = 15
sleep_time = 8
print(sleep_time > max_time)
derive = True
drink = False
both = drink and derive
print(both)
```



# Metodi

## Metodi di oggetti

- La maggior parte dei tipi di oggetti Python (a parte i numeri) possiede *metodi*
- Un metodo è semplicemente una funzione che esegue una certa operazione sull'oggetto
- Un metodo può avere degli argomenti e/o restituire un risultato (come abbiamo visto per le funzioni. e.g. `len("abcd")`)
- I metodi di un oggetto si chiamano tramite un riferimento all'oggetto stesso, facendolo seguire da “.” e il nome del metodo, eventualmente seguito dai valori degli argomenti tra parentesi

```
>>> s = "abcd"  
>>> s.replace("bc", "xx")  
'axxd'
```



# Metodi di stringhe

## Ricerche

`s.find(sub)` Restituisce l'indice della prima occorrenza della sottostringa `sub` (o -1 se `sub` non c'è in `s`). È possibile specificare una gamma di posizioni in cui cercare più piccola della stringa intera

```
>>> s = "abbcddbbc"  
>>> s.find("bc")  
2  
>>> s.find("bc", 4)  
7  
>>> s.find("bc", 4, 6)  
-1
```



# Metodi di stringhe

## Ricerche

`endswith` restuisce vero se la stringa finisce con una certa sottostringa (o una scelta da una tupla), falso altrimenti.

```
>>> s = "abbcddbbc"  
>>> s.endswith("bc")  
True  
>>> s.endswith("xx")  
False  
>>> s.endswith(("bb", "bc"))  
True
```



# Metodi di stringhe

## Modifiche

**replace** sostituisce tutte le occorrenze (o il numero richiesto) di una certa sottostringa con un'altra

```
>>> s = " MET CYS ASP CYS\n"
>>> s.replace("CYS", "C")
' MET C ASP C\n'
>>> s.replace("CYS", "C", 1)
' MET C ASP CYS\n'
```

**upper,lower** convertono una stringa in maiuscola o minuscola

```
>>> s.lower()
' met cys asp cys\n'
```

**lstrip,rstrip,strip** rimuovono spazi,tab e ritorni a capo da inizio, fine o da inizio e fine stringa

```
>>> s.lstrip()      >>> s.rstrip()
' MET CYS ASP CYS\n'     ' MET CYS ASP CYS'
>>> s.strip()
' MET CYS ASP CYS'
```



# Metodi di stringhe

## Conversioni con liste

`split` permette di convertire una stringa in una lista di sottostringhe, specificando il delimitatore da usare (spazi, tab o ritorni a capo di default)

```
>>> s = "spazi o virgole, a scelta, per separare"
>>> s.split()
['spazi', 'o', 'virgole,', 'a', 'scelta, ', 'per',
 'separare']
>>> s.split(",")
['spazi o virgole', ' a scelta', ' per separare']
```



# Metodi di stringhe

## Conversioni con liste

`join` permette di convertire una lista in una stringa, concatenando gli elementi e usando la stringa di partenza come delimitatore

```
>>> s = ":"  
>>> s.join(["a", "b", "c", "d"])  
'a:b:c:d'
```

## Nota

I metodi si possono applicare anche direttamente agli oggetti, non solo a variabili che siano loro riferimenti

```
>>> "".join(["a", "b", "c", "d"])  
'abcd'
```



# È il vostro turno!

- Use a variable to contain your favorite number, create a message that reveals your favorite number.
- Given a string, print it with capital letters. Then, capitalize only the first letter of each word.
- Print the length of the string.



# Control Flow and Branching

If – Else statements.

Loops



# If - Else Statements

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

- <condition> has a value True or False
- evaluate expressions in that block if <condition> is True



# Indentazione in Python

- Python uses indentation to indicate blocks, instead of { }
  - Makes the code simpler and more readable
  - In Java, indenting is optional. In Python, you **must** indent.



# Python loops

Python supporta due cicli differenti, ma la cui funzione ultima è quella di ripetere parti di codice fintanto che una o più condizioni sono soddisfatte.

1. Il **for loop**, più efficace quando si conosce esattamente il numero massimo di iterazioni. Utilizza nativamente una variabile di conteggio e può essere sempre scritto in una maniera alternativa utilizzando un while loop.
2. Il **while loop**, efficace quando non si conosce esattamente il numero massimo di iterazioni possibili. Può utilizzare una variabile di conteggio, ma ATTENZIONE AI LOOP INFINITI! Non sempre si è in grado di riscrivere un'iterazione col while loop in una che utilizzi il for loop.



# Esercizio

Scrivere un programma che chieda all'utente due numeri interi in e che controlli l'eventuale inserimento di quantità non numeriche.

1. Il programma deve riconoscere e stampare a video un messaggio che identifichi il numero più grande e più piccolo. Chiedi l'inserimento una sola volta. I due devono essere separati da spazi.
2. Se i due numeri sono uguali, il programma deve richiedere esplicitamente un nuovo numero diverso dai due precedenti. Poi effettuare il controllo al punto 1.
3. Se l'utente inserisce quantità non numeriche è necessario richiedere nuovamente l'inserimento.



Scrivi un programma che legga le ore lavorate da due impiegati e scriva il totale e la media giornaliera (per impiegato) ed il totale di ore lavorate.

- Riduci le ore inserite dall'utente ad un massimo di 8.

Employee 1: How many days? 3

Hours? 6

Hours? 12

Hours? 5

Employee 1's total hours = 19 (6.33 / day)

Employee 2: How many days? 2

Hours? 11

Hours? 6

Employee 2's total hours = 14 (7.00 / day)

Total hours for both = 33



Modificare il programma precedente affinchè si dia la possibilità all'utente di inserire il numero di impiegati totale che si vuole considerare.

- Per ogni impiegato, il programma deve chiedere di inserire quanti giorni ha lavorato e, per ogni giorno, chiedere di specificarti il numero di ore lavorate ogni giorno (8 ore al max).

**ATTENZIONE!** GLI STRAORDINARI NON SONO PAGATI E NON CONTANO DUNQUE NEL CONTEGGIO TOTALE DELLE ORE.

- Considerando che ogni impiegato guadagna 14 euro l'ora, calcolate quanto l'azienda deve pagare ciascuno.
- Tutte le informazioni riguardanti ogni impiegato, quali: ore medie, ore giornaliere, ore totali e stipendio, devono essere stampate al termine del programma.

➤ *Se l'utente cambia idea e decide di inserire meno impiegati di quelli che aveva previsto, il programma deve proseguire.*



# Gestione degli errori

Due approcci differenti: LBYL vs EAFP



# Gestione degli errori

Ci sono due strategie di gestione degli errori:

- **look before you leap** (LBYL): in questo caso si prevengono gli errori prima che questi accadano veramente. È il primo approccio alla gestione errori che è stato introdotto (vedi il linguaggio C).
- **Easier to ask forgiveness than permission** (EAFP): si permette che situazioni eccezionali accadano e, se avvengono, si gestiscono di conseguenza.



# LBYL vs EAFP

```
func SomeFunc(arg int) error {
    result, err := DoSomething(arg)
    if err != nil {
        // Handle the error here..
        log.Println(err)
        return err
    }
    return nil
}
```

**LBYL**



# LBYL vs EAFP

```
def to_integer(value):  
    try:  
        return int(vale)  
  
    except ValueError:  
        return None
```

EAFP



# LBYL vs EAFP

- Python supporta entrambe le due filosofie
- Sembra però che la community preferisca utilizzare maggiormente EAFP.
- Utilizzare un approccio piuttosto che un altro può avere impatti sul numero di check (if) da effettuare nel codice, sulle performance e sulla chiarezza e leggibilità del codice (quest'ultima molto importante).



# Operazioni Con e Su Liste



# Liste

- Collezioni **ordinate** di informazioni accessibili tramite un indice.
- Per accedere agli elementi si utilizzano le parentesi **quadre** [ ]
- È possibile salvare dati **omogenei** e **disomogenei**. Cercate di utilizzare liste di elementi dello stesso tipo se potete!
- Le liste sono oggetti **mutabili**. Potete espandere, ridurre e modificare gli elementi interni durante l'esecuzione del programma.

*Le liste sono usate frequentemente in tutte le applicazioni Python, proprio per le loro caratteristiche.*



# Liste

a\_list = [] empty list

L = [2, 'a', 4, [1, 2]]

len(L) → evaluates to 4

L[0] → evaluates to 2

L[2]+1 → evaluates to 5

L[3] → evaluates to [1, 2], another list!

L[4] → gives an error

i = 2

L[i-1] → evaluates to 'a' since L[1]='a' above



# Esercizio (1/2)..

- Dato un input di soli numeri, sostituisci ai numeri le seguenti lettere.

- 0 → O
- 1 → I
- 2 → D
- 3 → E
- 4 → A
- 5 → S
- 6 → G
- 7 → T
- 8 → B
- 9 → L



## Esercizio (2/2)..

- Se l'utente inserisce lettere, scarta le lettere non presenti al punto precedente. Sostituisci le rimanenti con i rispettivi numeri.
- Se l'utente inserisce sia numeri che lettere, scarta le lettere non presenti al punto 1. Sostituisci le lettere ai numeri e i numeri alle lettere.

**SE LE LETTERE SONO MINUSCOLE, METTILE MAIUSCOLE!**

Esempio:

Input: 0368

Output: OEGB

Input: gasatoq

Output: 645470

(la q deve essere ignorata)

Input: 33abc33

Output: EE48EE

(la c deve essere ignorata)



## CHANGING ELEMENTS

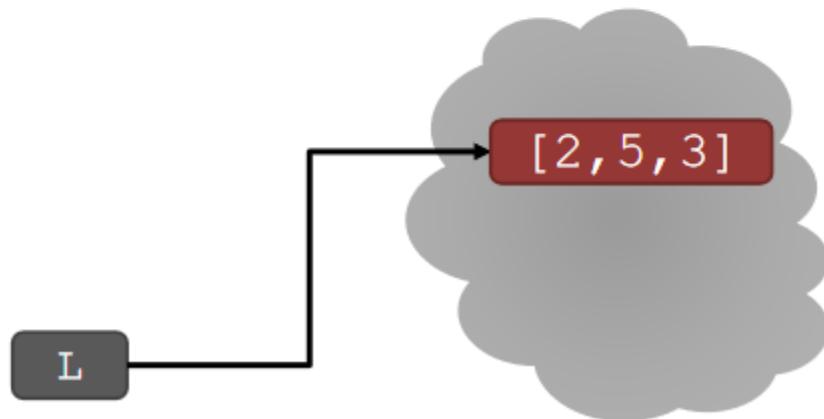
---

- lists are **mutable**!
- assigning to an element at an index changes the value

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

```
L[1] = 5
```

- L is now [2, 5, 3], note this is the **same object** L





# Liste

## Descrizione

- Una lista è una **sequenza** di oggetti qualunque (anche di tipo diverso, anche altre liste)

```
>>> l = ["AG01", 857, ["PAZ", "Piwi"]]
```

- Essendo una sequenza, condivide le operazioni su sequenza viste per le stringhe

```
>>> len(l)
```

```
3
```

- La lista è un tipo **mutabile**: può essere allungata ed accorciata, e si possono modificare i suoi elementi

```
>>> l[0] = "AGO1_HUMAN"
```

```
>>> l
```

```
["AGO1_HUMAN", 857, ["PAZ", "Piwi"]]
```



# Operazioni su liste

## Operatori

```
>>> l = ["AGO1", "AKAP1"]
>>> l + ["ACO1", "AGO3"]
["AGO1", "AKAP1", "ACO1", "AGO3"]

>>> l + []          # [] indica una lista vuota
["AGO1", "AKAP1"]

>>> l * 2
["AGO1", "AKAP1", "AGO1", "AKAP1"]

>>> "AKAP1" in l
True
>>> ["AGO1", "AKAP1"] in l # vale solo per singoli e
False
```



# Operazioni su liste

## Indicizzazione e sottolista

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[2]
3
>>> l[-1]
6
>>> l[:]
[1, 2, 3, 4, 5, 6]
>>> l[3:]
[4, 5, 6]
>>> l[:-2]
[1, 2, 3, 4]
```



# Iterare su liste

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
total = 0  
  
for i in range(len(L)):  
    total += L[i]  
  
print total
```

```
total = 0  
  
for i in L:  
    total += i  
  
print total
```

like strings,  
can iterate  
over list  
elements  
directly

- notice
  - list elements are indexed 0 to `len(L) - 1`
  - `range(n)` goes from 0 to `n-1`



# Operazioni su liste

## Matrici

- Una lista i cui elementi siano tutte liste implementa una *matrice*
- Si possono recuperare tramite uno o due indici righe o singoli elementi della matrice

```
>>> matrix = [[1,2,3], [4,5,6], [7,8,9]]  
>>> matrix[2]  
[7, 8, 9]  
>>> matrix[2][0]  
7  
>>> matrix[2][0:2]  
[7, 8]
```



# Modifica di liste

## Modifica con assegnazione ad indici

- Essendo oggetti mutevoli, è possibile modificare il contenuto di una lista

```
>>> l = [0, 0, 0, 0]
>>> p = l
>>> l[1] = 1
>>> l
[0, 1, 0, 0]
>>> p
[0, 1, 0, 0]
>> l[2] += 1
>>> p
[0, 1, 1, 0]
```

- Poiché l'oggetto cambia, tutte le variabili che vi si riferiscono “vedono” il cambiamento



# Modifica di liste

## Modifica tramite la funzione `del`

- La funzione `del` permette di cancellare elementi o sottoliste specificando posizione o gamma di posizioni

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> del(l[2])
>>> l
[1, 2, 4, 5, 6]
>>> del(l[-1])
>>> l
[1, 2, 4, 5]
>>> del(l[2:4])
>>> l
[1, 2]
>>> del(l[:])
>>> l
[]
```



# Metodi di lista

## Metodi di modifica

- Come i tipi stringa, anche le liste hanno una serie di metodi da eseguire
- Essendo oggetti mutabili, molti di questi metodi modificano direttamente l'oggetto

`append` aggiunge un elemento in fondo alla lista

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l.append(7)
>>> l
[1, 2, 3, 4, 5, 6, 7]
```

`extend` estende la lista attaccando in fondo tutti gli elementi di una lista

```
>>> l.extend([8, 9])
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```



# Metodi di lista

## Metodi di modifica

`insert` Inserisce un elemento prima di una certa posizione

```
>>> l = [1,2,3,4,5,6]
>>> l.insert(3,3.5)
>>> l
[1, 2, 3, 3.5, 4, 5, 6]
```



# Metodi di lista

## Metodi di modifica

**sort** Ordina la lista (assume l'esistenza di una funzione di ordinamento tra elementi, si usa per liste di elementi dello stesso tipo)

```
>>> l = ["basso", "medio", "alto"]
>>> l.sort()
>>> l
['alto', 'basso', 'medio']
```

**reverse** Riordina la lista dall'ultimo al primo elemento

```
>>> l.reverse()
>>> l
['medio', 'basso', 'alto']
```



# List comprehensions

## Operazioni sugli elementi di una lista

- Il costrutto nominato *list comprehension* è un tipo di espressione caratteristica del Python che permette di creare una lista come risultato di un'operazione da fare sugli elementi di un'altra lista

```
>>> L = [0,1,2,3,4]
>>> [i+1 for i in L]
[1, 2, 3, 4, 5]
```

- Il costrutto è composto da:
  - Un oggetto di tipo sequenza su cui iterare (e.g. la lista L)
  - Una variabile (o più, vedremo) che raccolga di volta in volta gli elementi (e.g. i)
  - Una espressione che faccia qualche operazione che coinvolga l'elemento cui la variabile si riferisce (e.g. i+1)
  - Le parole riservate for ed in, e le parentesi quadre a delimitazione



# List comprehensions

## Operazioni sugli elementi di una lista

- Ad esempio può essere usato per estrarre una colonna da una matrice:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
>>> [row[2] for row in matrix]  
[3, 6, 9]
```

- L'operazione sugli elementi può anche essere una qualsiasi funzione o metodo compatibile con il loro tipo

```
>>> [str.upper() for str in ["a", "b", "c"]]  
['A', 'B', 'C']
```

- E' possibile selezionare gli elementi della lista da processare tramite una condizione (parola chiave `if`)

```
>>> [i for i in [1, 3, 4, 5, 6, 2, 8, 9] if i % 2 != 0]  
[1, 3, 5, 9]      # solo numeri dispari
```



# List comprehensions

- Il costrutto può essere usato con oggetti che siano *sequenze*, tipo le stringhe!

```
[ char for char in 'AHHDCCEDGGTA' if char in 'HC' ]
```

- Può essere combinato con metodi o funzioni che processino liste.

```
''.join([char for char in 'AHHDCCEDGGTA' if char in  
         'HC' ])
```



# Mutation, Aliasing, Cloning

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object
- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**



# Mutation, Aliasing, Cloning

- attributes of a person
  - singer, rich
- he is known by many names
- all nicknames point to the **same person**
  - add new attribute to **one nickname** ...

Justin Bieber    singer    rich    troublemaker

- ... **all his nicknames** refer to old attributes AND all new ones

The Bieb	singer	rich	troublemaker
JBeebs	singer	rich	troublemaker

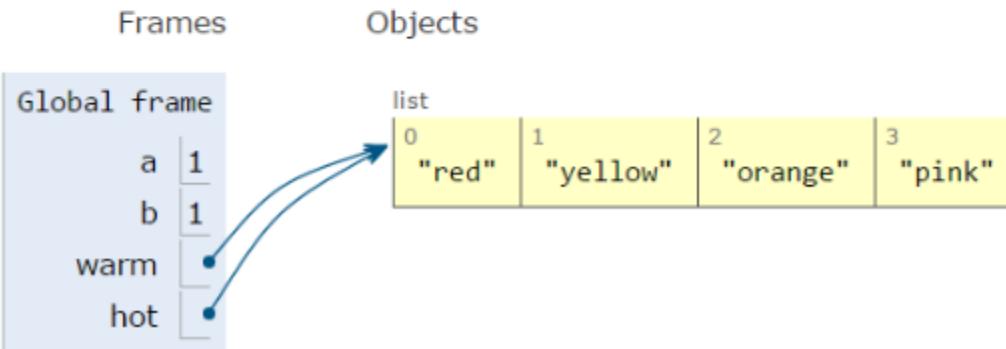


# Aliases

- hot is an **alias** for warm – changing one changes the other!
  - append () has a side effect

```
1 a = 1
2 b = a
3 print(a)
4 print(b)
5
6 warm = ['red', 'yellow', 'orange']
7 hot = warm
8 hot.append('pink')
9 print(hot)
10 print(warm)
```

```
| 1  
| 1  
| ['red', 'yellow', 'orange', 'pink']  
| ['red', 'yellow', 'orange', 'pink']
```



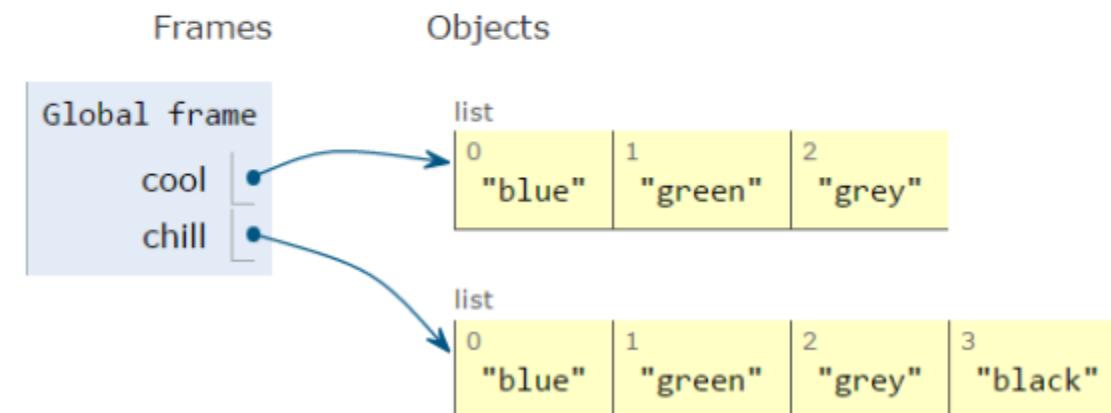


# Cloning a List

- create a new list and **copy every element** using  
`chill = cool[:]`

```
1 cool = ['blue', 'green', 'grey']
2 chill = cool[:]
3 chill.append('black')
4 print(chill)
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```





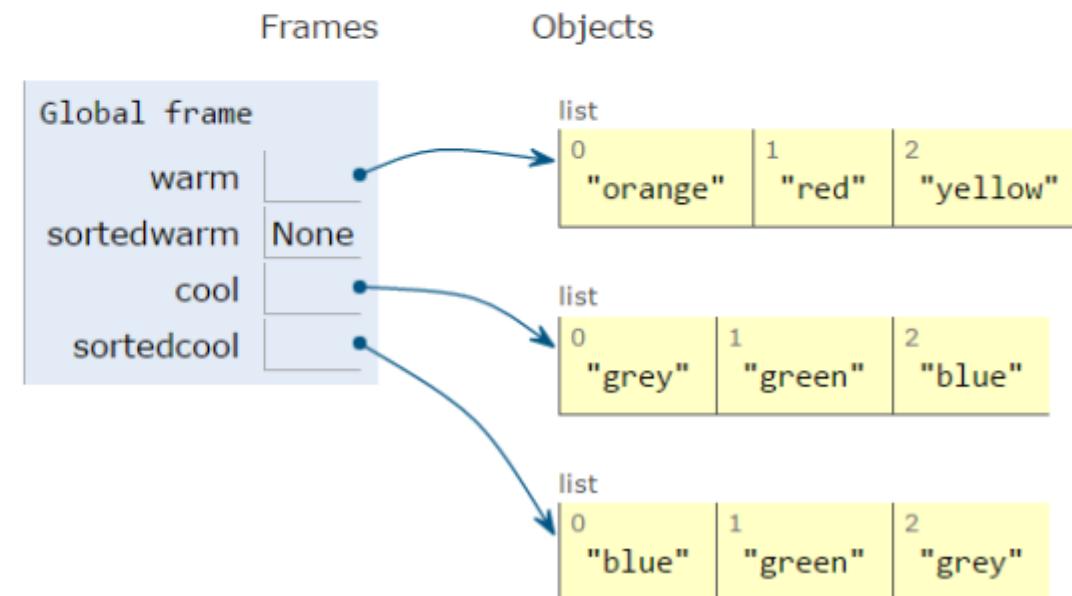
# Sorting

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

```
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

cool = ['grey', 'green', 'blue']
sortedcool = sorted(cool)
print(cool)
print(sortedcool)
```

```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```





# Converting strings into lists

- convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`
- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter
- use `' '.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

<code>s = "I&lt;3 cs"</code>	→ <code>s</code> is a string
<code>list(s)</code>	→ returns <code>['I', '&lt;', '3', ' ', 'c', 's']</code>
<code>s.split('&lt;')</code>	→ returns <code>['I', '3 cs']</code>
<code>L = ['a', 'b', 'c']</code>	→ <code>L</code> is a list
<code>' '.join(L)</code>	→ returns <code>"abc"</code>
<code>'_'.join(L)</code>	→ returns <code>"a_b_c"</code>



# Tuples



# Tuples

- an ordered sequence of elements, can mix element types
- cannot change element values, **immutable**
- represented with parentheses

remember  
strings?

te = () empty tuple

t = (2, "mit", 3)

t[0] → evaluates to 2

(2, "mit", 3) + (5, 6) → evaluates to (2, "mit", 3, 5, 6)

t[1:2] → slice tuple, evaluates to ("mit", )

extra comma  
means a tuple  
with one element

t[1:3] → slice tuple, evaluates to ("mit", 3)

len(t) → evaluates to 3

t[1] = 4 → gives error, can't modify object



# Tuple

## Descrizione

- Una tupla è una collezione ordinata di oggetti (racchiusi da parentesi tonde)

```
>>> t = ("C", 34)
```

- Come stringhe e liste, è un tipo *sequenza*, accessibile per indice e su cui funzionano le operazioni definite su sequenze

```
>>> t[1]
```

```
34
```

- Come le stringhe, è *immutabile* e non è possibile modificarla

```
>>> t[1] += 1
```

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

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

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



# Operazioni su tuple

## Operatori

```
>>> t = (1, -5, 4)
>>> t + (1, 2)
(1, -5, 4, 1, 2)
>>> t + ()
(1, -5, 4)
>>> t * 2
(1, -5, 4, 1, -5, 4)
```



# Operazioni su tuple

## Tuple con un solo elemento

```
>>> t = (1, -5, 4)
>>> t + (1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple
              (not "int") to tuple
>>> t + (1,)
(1, -5, 4, 1)
```

## Nota

- Anche un'espressione può essere racchiusa tra parentesi
- Per distinguere una tupla con un solo elemento da un'espressione, va aggiunta una virgola dopo l'elemento (e.g. (1,))



# Operazioni su tuple

## Indicizzazione e sottotupla

```
>>> t = (1, 2, 3, 4, 5, 6)
>>> t[2]
3
>>> t[-1]
6
>>> t[:]
(1, 2, 3, 4, 5, 6)
>>> t[3:]
(4, 5, 6)
>>> t[:-2]
(1, 2, 3, 4)
```



# Immutabilità di tuple

- Come le stringhe, le tuple sono oggetti immutabili

```
>>> t = (12, "abc", [1,2,3])
>>> t[2] = "f"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
                        assignment
```

- Tale immutabilità non si applica ai contenuti della tupla, per cui i suoi oggetti possono essere modificati *se mutabili*

```
>>> t[2].append(4)
>>> t
(12, 'abc', [1, 2, 3, 4])
>>> t[1][2] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
                        assignment
```



# Perché le tuple?

## Utilità delle tuple

- Le tuple sono semplicemente liste immutabili
- Non hanno quindi nessun metodo (ma possono essere manipolate per creare nuove tuple con gli operatori)
- La loro utilità è data dal fatto di essere immutabili: possono essere passate tra parti diverse di un programma essendo sicuri che non verranno modificate (e.g. in una certa posizione ci sarà sempre lo stesso oggetto)
- Possono essere usate dove c'è bisogno di oggetti immutabili, ad esempio come indici di dizionari



# Perché le tuple?

- conveniently used to **swap** variable values

x = y

y = x



temp = x

x = y

y = temp



(x, y) = (y, x)



- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):  
    q = x // y  
    r = x % y  
    return (q, r)
```

integer  
division

```
(quot, rem) = quotient_and_remainder(4, 5)
```



# Perché le tuple?

## Utilità delle tuple

- Le tuple sono spesso usate per semplificare le operazioni di assegnazione a più variabili

```
>>> (x, y, z) = (1.2, 3, 4.5)  
>>> y  
3
```

- Dove non ambiguo, è possibile specificare una tupla anche senza parentesi, semplificando ulteriormente la notazione

```
>>> x, y, z = 1.2, 3, 4.5  
>>> y, z  
(3, 4.5)
```



# Perché le tuple?

## Utilità delle tuple

- Vari metodi e funzioni restituiscono tuple in uscita, ad esempio il metodo `items` dei dizionari

```
>>> D = { "C" : 8, "H" : 12}
>>> D.items()
dict_items([('C', 8), ('H', 12)])
```

- Il costrutto list comprehension può gestire tuple di variabili invece di variabili singole:

```
>>> [ "%s = %d" % (k,v) for (k,v) in
... D.items() ]
['C = 8', 'H = 12']
```



# Dictionaries



# Dizionari

- I dizionari sono collezioni **non ordinate** di oggetti (**items**) indirizzati attraverso chiavi identificative (**keys**).
- I dizionari sono quindi utili per connettere tra loro informazioni correlate attraverso l'associazione chiave – valore.
- Una volta che avete dimestichezza potete modellare una moltitudine di oggetti o situazioni reali



# Come salvare le informazioni di uno studente?

- so far, can store using separate lists for every info

```
names = ['Ana', 'John', 'Denise', 'Katy']
grade = ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person



# Come estrarre efficacemente?

```
i = name_list.index(student)  
grade = grade_list[i]  
course = course_list[i]  
return (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists



# Un approccio diverso..

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
...	...

index      element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
...	...

custom  
index by  
label      element



# Implementazione a dizionario

- store pairs of data
  - key
  - value

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

my\_dict = `{}` *empty dictionary*

grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'Katy': 'A'}

↑      ↑      ↑      ↑      ↑      ↑      ↑      ↑  
key1 val1    key2 val2    key3 val3    key4 val4

*custom  
index by  
label*

*element*



# Implementazione a dizionario

- similar to indexing into a list
- **looks up the key**
- **returns the value** associated with the key
- if key isn't found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}  
grades['John']      → evaluates to 'A+'  
grades['Sylvan']    → gives a KeyError
```



# Implementazione a dizionario

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

'John' in grades	→ returns True
'Daniel' in grades	→ returns False

- **delete** entry

```
del(grades['Ana'])
```

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'



# Implementazione a dizionario

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- get an **iterable that acts like a tuple of all keys**

grades.keys() → returns ['Denise', 'Katy', 'John', 'Ana']  
*no guaranteed order*

- get an **iterable that acts like a tuple of all values**

grades.values() → returns ['A', 'A', 'A+', 'B']  
*no guaranteed order*

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'



# Riassumendo..

- values
  - any type (**immutable and mutable**)
  - can be **duplicates**
  - dictionary values can be lists, even other dictionaries!
- keys
  - must be **unique**
  - **immutable** type (int, float, string, tuple, bool)
    - actually need an object that is **hashable**, but think of as immutable as all immutable types are hashable
  - careful with **float** type as a key
- **no order** to keys or values!

```
d = {4:{1:0}, (1,3) :"twelve", 'const':[3.14,2.7,8.44]}
```



# Dizionari

## Creazione

- Un dizionario può essere creato specificandone l'insieme di coppie chiave:valore separate da virgole e delimitato da parentesi graffe

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
```

- E' comune creare un dizionario vuoto per poi riempirlo inserendo coppie di volta in volta

```
>>> D = {}
```



# Dizionari

## Dizionari sparsi

Le chiavi di un dizionario non devono necessariamente essere stringhe, basta che siano oggetti *immutabili* (e.g. numeri o tuple). E' così possibile creare ad esempio vettori o array multidimensionali *sparsi* (con pochi indici con valore specificato, assumendo ad es. una valore di default come 0 per gli altri)

```
>>> D = {10 : 2.3, 50 : 1.5, 223 : -3}
>>> D = { (0,0) : "rosso", (0,1) : "verde" ,
... (100,200) : "blu"}
```



# Dizionari

## Ricerca

- Verifica della presenza di una certa chiave (operatore `in`)

```
>>> "alto" in D
```

```
True
```

```
>>> "altissimo" in D
```

```
False
```

- metodo `get`: recupero di un oggetto specificando chiave e valore di default da restituire se non presente (`None` se non specificato)

```
>>> D.get("alto")
```

```
2
```

```
>>> D.get("altissimo") # None non viene stampato
```

```
>>> D.get("altissimo", "alto")
```

```
'alto'
```



# Dizionari

## Metodi di esplorazione

**len** restituisce il numero di elementi (coppie chiave, valore) contenuti nel dizionario

```
>>> D = {"a" : 0, "b" : 1, "c" : 2}  
>>> len(D)  
3
```

**keys** restituisce una vista sulle chiavi nel dizionario

```
>>> D.keys()  
dict_keys(['a', 'c', 'b'])
```

**values** restituisce una vista sui valori nel dizionario

```
>>> D.values()  
dict_values([0, 2, 1])
```

**items** restituisce una vista delle coppie chiave, valore (come tuple) nel dizionario

```
>>> D.items()  
dict_items([('a', 0), ('c', 2), ('b', 1)])
```



# Modifica di dizionari

## Oggetti mutabili

- Il modo più semplice per recuperare un oggetto è accedendovi per chiave (come si farebbe in una lista con l'indice)

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}  
>>> D["alto"]  
2
```

- Come le liste, i dizionari sono oggetti *mutabili*. E' quindi modificare un oggetto associato ad una certa chiave

```
>>> D["alto"] = 3  
>>> D  
{'alto': 3, 'basso': 0, 'medio': 1}
```



# Modifica di dizionari

## Oggetti mutabili

- A differenza delle liste se si assegna un valore ad una chiave non presente, la coppia `chiave:valore` viene aggiunta al dizionario:

```
>>> D["altissimo"] = 3  
>>> D  
{'alto': 3, 'basso': 0, 'altissimo': 3, 'medio': 1}
```

- In una lista, l'assegnazione di un valore ad un indice fuori dimensione genera invece un errore (si deve usare `append`)

```
>>> L = ["a", "b", "c"]  
>>> L[3] = "d"  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: list assignment index out of range
```



# Modifica di dizionari

## Modifica tramite la funzione `del`

- In modo analogo alle liste, la funzione `del` permette di cancellare elementi (ma non sottoinsiemi di elementi) specificandone la chiave

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}  
>>> del(D["medio"])  
>>> D  
{'basso': 0, 'alto': 2}
```



# Comando del

## Rimozione di variabili

- La funzione `del` applicata ad una variabile la elimina, ma non cancella l'oggetto a cui si riferisce

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
>>> D2 = D
>>> del(D)
>>> D
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'D' is not defined
>>> D2
{"basso" : 0, "medio" : 1, "alto" : 2}
```



# Ora è il vostro turno

- Usate un dizionario per salvare informazioni riguardanti persone che conoscete. Cominciate con il nome, cognome, età, altezza, tipo di lavoro/studio e la città di residenza. Stampate il resoconto di ogni persona.
- Usate lo stesso dizionario per aggiungere il numero di telefono. Utilizzate veri numeri se volete!
- Usate un dizionario per realizzare un glossario. Come key utilizzate parole nuove che avete imparato in queste ultime settimane (anche in questo corso!) e come item la definizione. Stampate a video con una formattazione leggibile, ogni voce deve essere separata da una linea vuota.



# Sets



# Sets

## Descrizione

- Un set è una collezione non ordinata di oggetti.  

```
>>> s = {"1dsf", "3rsf", "4rews", "1ewae"}
```
- È come un dizionario che contenga solo chiavi e non valori
- Lo stesso oggetto non può comparire più di una volta, per cui creare un set da un elenco equivale ad eliminarne i duplicati



# Sets

## Ricerca

Si usa l'operatore in

```
>>> S = set(["1dsf", "3rsf", "1dsf", "4rews", "3rsf"])
>>> "3rsf" in S
True
```

## Inserimento

- Metodo `add` per aggiungere singoli elementi

```
>>> S = set(["1dsf", "3rsf", "1dsf", "4rews", "3rsf"])
>>> S.add("2rsa")
>>> S
{'3rsf', '1dsf', '4rews', '2rsa'}
```

- Metodo `update` per aggiungere elenchi di elementi

```
>>> S.update(["1aaa", "1asd", "1aaa"])
>>> S
{'2rsa', '4rews', '1aaa', '3rsf', '1asd', '1dsf'}
```



# Sets

## Creazione

- Un set può essere creato come elenco di oggetti tra parentesi graffe

```
>>> S = {"1dsf", "3rsf", "4rews", "1ewae"}
```

- Un set può essere creato passando una lista alla funzione `set`

```
>>> S = set(["1dsf", "3rsf", "1dsf", "4rews", "3rsf"])
>>> S
{'4rews', '1dsf', '3rsf'}
```

- Un set vuoto può essere creato usando `set` senza argomenti (`{}` crea un dizionario):

```
>>> S = set()
```



# Gestione dei files



# Il concetto di file

- Un file è una *astrazione fornita dal sistema operativo* il cui scopo è consentire la memorizzazione di informazioni contenute in un moderno computer.
- Un file può essere inteso come un contenitore di dati archiviati sottoforma di sequenze di bytes.
- All'interno di un computer i files sono tipicamente salvati nel dispositivo di archiviazione di massa, quale HDD o SSD.
- Un file può avere una qualsiasi estensione. Alcuni esempi sono *.py, .pdf, .docx*.



# Il concetto di file

- Per lavorare con i file è necessario utilizzare un oggetto speciale di tipo file, chiamato *file object* o *file-like object*.
- Il *file object* fornisce una interfaccia per scrivere, leggere file in memoria.
- Ci sono due tipologie di files: *binary files*, *text files*. Ci occuperemo dei file di testo principalmente.
- Un file può avere una qualsiasi estensione. Alcuni esempi sono *.py*, *.pdf*, *.docx*.
- **In Python è molto semplice manipolare files!**



# Files

## Creazione

- Un oggetto file si crea apendo un file
- La funzione `open` prende come argomenti:
  - il percorso nel file system del file da aprire (assoluto o relativo alla directory corrente)
  - La modalità con cui aprire il file, tra cui:
    - `r` lettura (il file deve esistere già )
    - `w` scrittura (il file viene sovrascritto, o creato se non esiste)
    - `a` estensione (se il file è già esistente, l'output viene aggiunto in fondo)
  - aggiungendo un `+` ad una delle modalità , si apre il file sia in lettura sia in scrittura

```
>>> f = open('/tmp/prova', "w+") scrittura e lettura  
>>> f = open('TODO') # r se non specificata
```



# Lettura e Scrittura di files

- Il metodo del *file object* `read()` legge l'intero contenuto del file e lo restituisce in una **unica** stringa.
- Il metodo del *file object* `readline()` legge l'intera riga, insieme al ritorno a capo, contenuta nel file e la restituisce in una **unica** stringa.
- Il metodo del *file object* `readlines()` legge l'intero contenuto del file e lo restituisce in una **lista** di stringhe, ciascuna contenente una riga e il ritorno a capo.



# Lettura e Scrittura di files

- Il metodo del *file object* `write()` permette di scrivere del contenuto nel file e restituisce il numero di caratteri effettivamente scritti sul file. **Il ritorno a capo conta come un carattere!**
- Il metodo del *file object* `writelines()` scrive tutte le stringhe della lista nel file. Il metodo non aggiunge il ritorno a capo, ci deve già essere e **non** restituisce nessun valore.
- **Ricordatevi di chiudere il file quando non vi serve più attraverso il metodo `close()`.**



# Chiusura di files

- Una volta che le operazioni sul file sono terminate è **obbligatorio** chiudere il file.
- Python garantisce la corretta gestione del file **solo dopo** che questo è stato chiuso.
- Se vi dimenticate di chiuderlo prima che il programma termini, tutti o parte dei cambiamenti effettuati potrebbero andare persi.
- **Python vi aiuta a non dimenticarvelo** tramite l'istruzione `with`.



# Files

## Scrivere e leggere oggetti

- I metodi visti scrivono e leggono stringhe (o liste di stringhe)
- Per poter scrivere e leggere oggetti diversi, devono essere convertiti in stringhe, e viceversa.
- Tale operazione può essere fatta in modi diversi, e il più conveniente dipende dagli oggetti in questione

```
>>> s = "same old string"
>>> x, y, z = -1, 4.5, 0.2
>>> l = [0, 1, 2, 3, 4]
>>> d = {3 : 1, 5 : 1} # vettore sparso
>>> f = open("data", "w")
```



# The import statement



# How import works

- Python tratta tutti i file con estensione .py come **moduli** software.
- I moduli possono essere importati all'interno di altri moduli e rappresentano dunque un primo strumento di organizzazione software.
- Potete implementare classi e funzioni all'interno di un modulo functions.py, importarlo nel vostro program.py e utilizzarle senza problemi.
- **Quando importate un modulo, Python lo eseguirà!** Una buona abitudine è isolare sempre il codice che non si vuole eseguire nella fase di import del modulo predisponendo un **entry point**.



# The import statement

- Ci sono vari modi con cui è possibile importare un modulo.
- Alcuni sono sconsigliabili e sono dunque segnati come **Very Bad Practice**. Quali tra questi secondo voi?

```
import Imports1  
  
Imports1.bar()  
Imports1.foo()  
Imports1.Color()
```

```
from Imports1 import foo, bar, open  
foo()  
bar()
```

```
import Imports1 as ip  
ip.foo()  
ip.bar()
```

```
from Imports1 import *
```