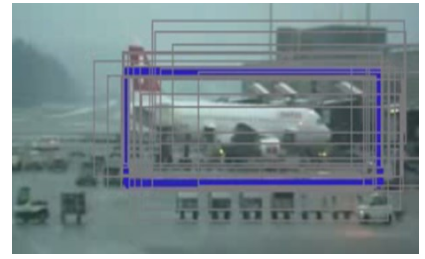


UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

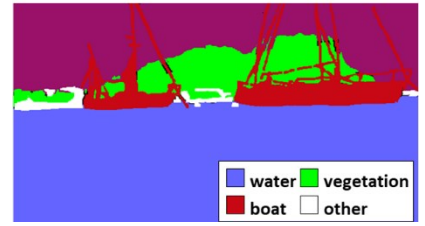
Corso di Sistemi Informativi
A.A. 2018/19

Docente
Domenico Daniele Bloisi

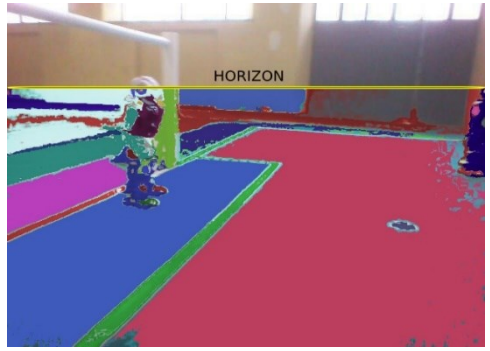




Dizionari e File in Python

Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)

Marzo 2019

Il corso

- Home page del corso
  https://dbloisi.github.io/corsi/sistemi-informativi.html

- Docente: Domenico Daniele Bloisi

- Periodo: II semestre marzo 2019 – giugno 2019

  Martedì 17:00-19:00 (Aula GUGLIELMINI)
  Mercoledì 8:30-10:30 (Aula GUGLIELMINI)

Obiettivi del corso

Il corso intende fornire agli studenti conoscenze relative alla programmazione in Python per lo sviluppo di applicazioni basate sul sistema operativo ROS, sulla libreria per la percezione OpenCV e sulla libreria per il Deep Learning Keras

https://www.youtube.com/watch?v=l9KYJlLnEbw

Tipi di dato strutturati di Python

Oltre alle stringhe, due dei principali tipi strutturati del linguaggio Python sono:

1. le liste, che consentono di rappresentare sequenze ordinate di valori qualsiasi
2. i dizionari, che consentono di rappresentare collezioni (non ordinate) di valori qualsiasi

Esistono anche altri tipi di dato strutturati (come le tuple) ed è possibile definirne di nuovi.

Tipo di dato dizionario

Le liste consentono di rappresentare dati strutturati i cui valori siano composti da una sequenza ordinata di valori più semplici.

Un altro caso comune nella pratica è quello in cui i valori dei dati da elaborare siano rappresentabili come collezioni (insiemi) di valori più semplici e non ordinati, ciascuno dei quali abbia un significato che possa essere descritto con un nome simbolico.
Un esempio sono le informazioni anagrafiche su uno studente: nome, cognome, data e luogo di nascita, codice fiscale, matricola, ecc.

I dati aventi tali caratteristiche possono essere rappresentati in Python per mezzo del tipo strutturato dizionario.

Tipo di dato dizionario

Il tipo di dato dizionario è costituito da collezioni

- non ordinate
- modificabili
- indicizzate

aventi di un numero qualsiasi di valori, ciascuno dei quali può appartenere a qualsiasi tipo di dato

Ognuno di tali valori è associato ad un identificatore univoco detto chiave, che di norma è una stringa, che deve essere scelto dal programmatore.

Tipo di dato dizionario

Come per le variabili e i nomi delle funzioni, anche per le chiavi dei dizionari è buona norma usare stringhe mnemoniche.

Un dizionario è quindi un insieme di coppie chiave/valore. In particolare, all'interno di un dizionario:

- non possono esistere chiavi identiche
- ogni elemento (coppia chiave/valore) è identificato univocamente dalla sua chiave
- tra gli elementi non è definito alcun ordinamento

Tipo di dato dizionario: sintassi

Un dizionario si rappresenta nei programmi Python come una sequenza di elementi

- ✓ racchiusi tra parentesi graffe
- ✓ separati da virgole

Ogni elemento è costituito da una coppia chiave/valore, in cui i due elementi di ogni coppia sono separati da due punti

```
{chiave1:valore1, chiave2:valore2, . . . }
```

Tipo di dato dizionario: esempi

In un dizionario che contiene informazioni anagrafiche su una persona relative a nome e cognome (i cui valori sono stringhe) ed età (il cui valore è un numero intero), le chiavi saranno le stringhe "nome", "cognome", "età":

```
{"nome":"Maria", "cognome":"Bianchi", "eta":28}
```

Per un dizionario che contiene le coordinate di un punto (due numeri frazionari) nel piano cartesiano, associate alle chiavi ‘x’ e ’y’, avremo:

```
{'x':-1.2, 'y':3.4}
```

Un dizionario che contiene una data (giorno, mese e anno, in formato numerico):

```
{"giorno":1, "mese":6, "anno": 2017}
```

un dizionario vuoto:

```
{}
```

Il tipo di dato dictionary

Anche i valori di tipo dizionario costituiscono espressioni Python.

È quindi possibile:

- stampare un dizionario con l'istruzione `print`, per esempio:

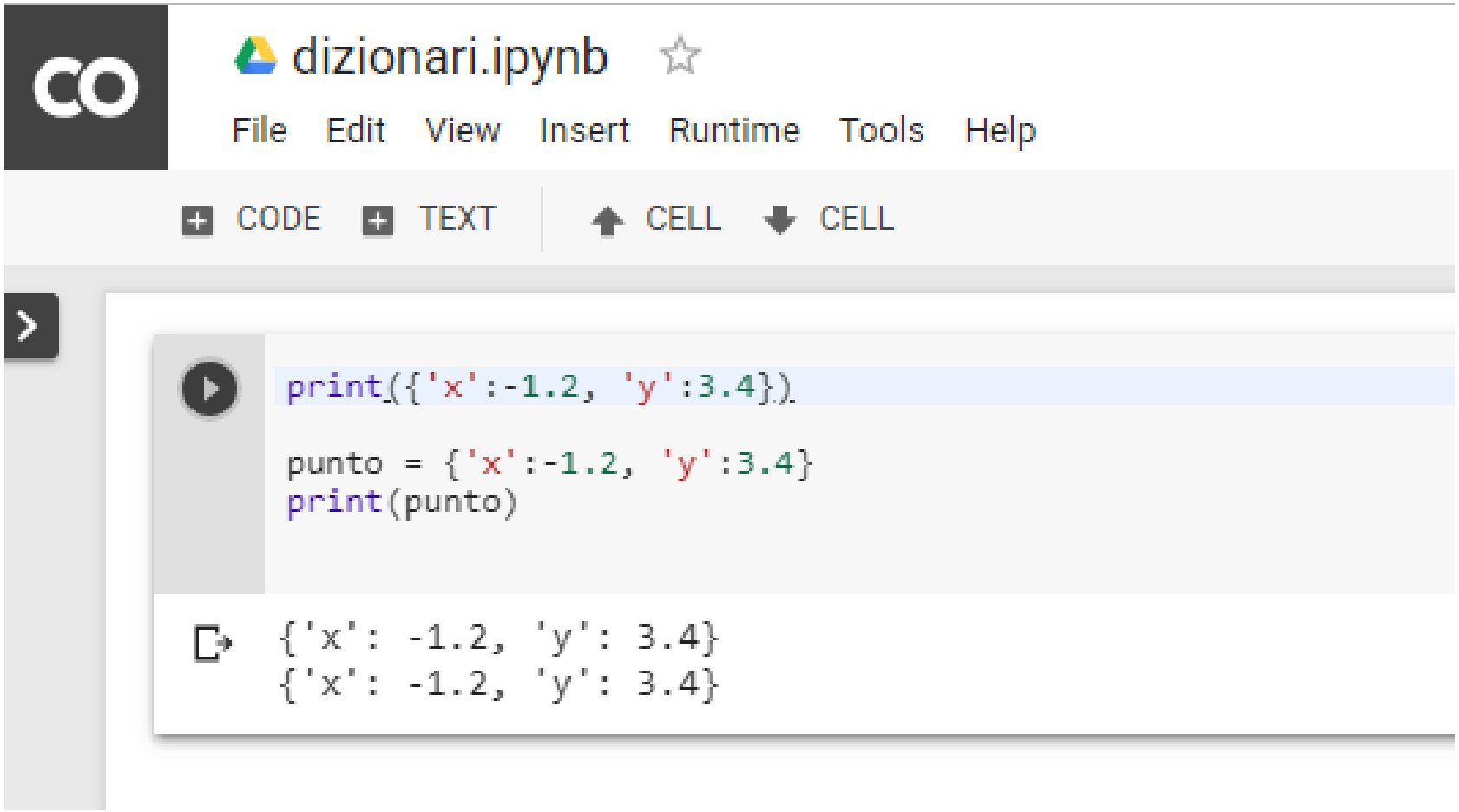
```
print({'x':-1.2, 'y':3.4})
```

- assegnare un dizionario a una variabile, per esempio:

```
punto = {'x':-1.2, 'y':3.4}
```

Nota: quando un dizionario viene stampato, le coppie chiave/valore compaiono in un ordine che non può essere controllato dal programmatore.

Il tipo di dato dictionary



Espressioni in dizionari

I valori di ogni elemento di un dizionario possono essere indicati attraverso espressioni.

Per esempio, dopo l'esecuzione della seguente sequenza di istruzioni:

```
n = "Maria"
c1 = "Ros"
c2 = "si"
a = 5
persona = {"nome": n, "cognome": c1 + c2, "eta": a ** 2}
```

la variabile persona sarà associata al seguente dizionario:

```
persona = {"nome":"Maria", "cognome":"Rossi", "eta":25}
```

Espressioni in dizionari

```
n = "Maria"
c1 = "Ros"
c2 = "si"
a = 5
persona = {"nome": n, "cognome": c1 + c2, "eta": a ** 2}

print(persona)
```

```
{'nome': 'Maria', 'cognome': 'Rossi', 'eta': 25}
```

Dizionari annidati

Come per le liste, anche gli elementi di un dizionario possono contenere valori di tipo qualsiasi e quindi anche valori strutturati come stringhe, liste e dizionari (annidati).

Per esempio, il dizionario seguente (non annidato) contiene il nome, il cognome e la data di nascita di una persona (giorno, mese e anno in formato numerico):

```
{"nome":"Ugo", "cognome":"Neri", 'g':1, 'm':1, "a":1995}
```

Le stesse informazioni possono essere memorizzate in due dizionari annidati:

```
{"nome":"Ugo", "cognome":"Neri", \
"data_nascita": {'g':1, 'm':1, "a":1995}}
```

Dizionari annidati

```
d1 = {"nome":"Ugo", "cognome":"Neri", 'g':1, 'm':1, "a":1995}
print(d1)

d2 = {"nome":"Ugo", "cognome":"Neri", \
"data_nascita": {'g':1, 'm':1, "a":1995}}
print(d2)
```

```
{'nome': 'Ugo', 'cognome': 'Neri', 'g': 1, 'm': 1, 'a': 1995}
{'nome': 'Ugo', 'cognome': 'Neri', 'data_nascita': {'g': 1, 'm': 1, 'a': 1995}}
```

Principali operatori per i dizionari

Sintassi	Descrizione
`dizionario1 == dizionario2`	confronto (“uguale a”)
`dizionario1 != dizionario2`	confronto (“diverso da”)
`dizionario[chiave]`	indicizzazione: accesso ai singoli elementi

Operatori di confronto

Gli operatori `==` e `!=` consentono di scrivere espressioni condizionali (il cui valore sarà `True` o `False`) consistenti nel confronto tra due dizionari.

Sintassi:

```
dizionario_1 == dizionario_2
dizionario_1 != dizionario_2
```

dove `dizionario_1` e `dizionario_2` indicano espressioni aventi come valore un dizionario.

Semantica:

due dizionari sono considerati identici se contengono le stesse coppie chiave/valore, indipendentemente dal loro ordine.

Operatori di confronto

```
match1 = {"team1":"Atletico Madrid", "team2":"Jueventus", "result":"2-0"}
match2 = {"team1":"Jueventus", "team2":"Atletico Madrid", "result":"3-0"}
print(match1 == match2)
print(match1 != match2)
match3 = {"result":"2-0", "team2":"Jueventus", "team1":"Atletico Madrid"}
print(match1 == match3)
```

```
False
True
True
```

L’operatore di indicizzazione

L'operatore di indicizzazione consente di accedere al valore di ogni elemento di un dizionario, per mezzo della chiave corrispondente.

Sintassi:
`dizionario[chiave]`

- `dizionario` deve essere una espressione che restituisce un dizionario
- `chiave` deve essere una espressione il cui valore corrisponde a una delle chiavi del dizionario

Semantica:
viene restituito il valore dell'elemento di `dizionario` associato a `chiave`. Se il valore di `chiave` non corrisponde ad alcuna delle chiavi di `dizionario`, si otterrà un messaggio di errore

L’operatore di indicizzazione

```
studente = {'nome':'Domenico', 'cognome':'Bloisi', 'matricola':110930}
print(studente['matricola'])
print(studente[matricola])
```

```
110930
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-76e87cd1264b> in <module>()
      1 studente = {'nome':'Domenico', 'cognome':'Bloisi', 'matricola':110930}
      2 print(studente['matricola'])
----> 3 print(studente[matricola])

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

SEARCH STACK OVERFLOW

L’operatore di indicizzazione

```
m = 'matricola'
studente = {'nome':'Domenico', 'cognome':'Bloisi', m:110930}
print(studente['matricola'])
print(studente[m])
```

```
110930
110930
```

Modificare un dizionario

L'operatore di indicizzazione consente anche di:

- modificare il valore associato a una chiave esistente
- aggiungere una nuova coppia `chiave:valore` se `chiave` non è presente nel dizionario

Sintassi:

```
dizionario[chiave] = valore
```

Semantica:

se `chiave` fa parte di dizionario, il valore precedente viene sostituito da `valore`; altrimenti viene aggiunto al dizionario il nuovo elemento `chiave:valore`

Modificare un dizionario

```
studente = {'nome':'Domenico', 'cognome':'Bloisi', 'matricola':110930}
print(studente)
studente['eta'] = 37
print(studente)
studente['matricola'] = 100100
print(studente)
```

```
{'nome': 'Domenico', 'cognome': 'Bloisi', 'matricola': 110930}
{'nome': 'Domenico', 'cognome': 'Bloisi', 'matricola': 110930, 'eta': 37}
{'nome': 'Domenico', 'cognome': 'Bloisi', 'matricola': 100100, 'eta': 37}
```

Esempio: costruzione di un dizionario

```
persona = {}
print ("Inserire i seguenti dati anagrafici:" )
persona["nome"] = input("Nome: ")
persona["cognome"] = input("Cognome: ")
persona["eta"] = input("Età: ")
print ("I dati inseriti sono:", persona)
```

```
Inserire i seguenti dati anagrafici:
Nome: Domenico
Cognome: Bloisi
Età: 37
I dati inseriti sono: {'nome': 'Domenico', 'cognome': 'Bloisi', 'eta': '37'}
```

Esempio: stampa di un dizionario

```python
def stampa_persona(p):
    print ("I dati anagrafici sono i seguenti:")
    print ("Nome:", p["nome"])
    print ("Cognome:", p["cognome"])
    print ("Età:", p["eta"])

persona = {}
print ("Inserire i seguenti dati anagrafici:" )
persona["nome"] = input("Nome: ")
persona["cognome"] = input("Cognome: ")
persona["eta"] = input("Età: ")
stampa_persona(persona)
```

```
Inserire i seguenti dati anagrafici:
Nome: Domenico
Cognome: Bloisi
Età: 37
I dati anagrafici sono i seguenti:
Nome: Domenico
Cognome: Bloisi
Età: 37
```

Liste di dizionari

```
rubrica = []
contatto1 = {'nome':'Nicola', 'numero':'328456723'}
contatto2 = {'nome':'Lucia', 'numero':'339474529'}
rubrica += [contatto1]
rubrica += [contatto2]
print(rubrica)
print(rubrica[1])
print(rubrica[1]['nome'])
```

```
[{'nome': 'Nicola', 'numero': '328456723'}, {'nome': 'Lucia', 'numero': '339474529'}]
{'nome': 'Lucia', 'numero': '339474529'}
Lucia
```

Liste vs dizionari

Si considerino le due alternative seguenti:

```
persona_a = {"nome":"Ada", "cognome":"Neri", "eta":25}
persona_b = ["Ada", "Neri", 25]
```

Per accedere al cognome di una persona l'espressione

```
persona_a["cognome"]
```

risulta certamente più comprensibile rispetto a

```
persona_b[1]
```

Liste vs dizionari

Si consideri ora il caso in cui si debba memorizzare una sequenza o una collezione (non ordinata) di valori, la cui dimensione non sia nota al programmatore **nel momento in cui scrive il programma**.

Per esempio, questo può accadere in un programma che debba elaborare i dati relativi a:

- un insieme di studenti che abbiano sostenuto un esame
- un insieme di atleti che abbiano partecipato a una gara

In questo caso è preferibile usare una lista, anche se tra i valori che si devono elaborare **non** esiste alcun ordinamento predefinito.

L'uso di un dizionario richiederebbe infatti al programmatore la scelta di una chiave distinta per ciascun valore da memorizzare al suo interno.

Liste vs dizionari

Si consideri ancora il caso dei dati su un esame sostenuto da un insieme di studenti.

Se si volessero memorizzare i dati di ogni studente come elementi di un dizionario (a loro volta contenuti in un dizionario con chiavi come "nome", "matricola", ecc.), si dovrebbero usare chiavi come "studente1", "studente2", ecc.
Per esempio:

```
{"studente1": {"matricola":"12345", "nome": "Luca", ... },
"studente2": {"matricola":"54321", "nome": "Ugo", ... } }
```

Se l'intero dizionario fosse memorizzato in una variabile di nome studenti, i dati di ogni studente sarebbero accessibili con una sintassi come

```
studenti["studente2"]
```

Questa sintassi non è però più mnemonica di quella richiesta per l'accesso agli elementi di una lista, per esempio

```
studenti[1]
```

Insiemi

Omettendo i valori delle chiavi in un dizionario si ottiene un insieme, i cui elementi possono solo essere oggetti immutabili.

Esempio

```
I = {0,2,4,6,8}
```

- Possiamo verificare con `in` e `not in` se un oggetto appartenga o meno a un insieme e anche scandirne gli elementi con un ciclo `for`
- Non possiamo associare a una chiave alcun valore, cioè `I[0]` darà luogo a un errore. Si noti che un valore ripetuto in un insieme conta una sola volta e che l'ordine è indefinito:

```
{1,1,1} == {1} → True
{1,2,3} == {2,1,3} → True
```

Insiemi

```
I = {0,2,4,6,8}
print(I[0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-3ec6bcc01974> in <module>()
      1 I = {0,2,4,6,8}
----> 2 print(I[0])

TypeError: 'set' object does not support indexing
```

SEARCH STACK OVERFLOW

Non è possibile accedere agli elementi di un set utilizzando l'indicizzazione poichè i set sono oggetti non ordinati e i suoi elementi non sono indicizzati.

Insiemi

Se `I` è un insieme possiamo:

- Aggiungere ad esso un elemento
  `I.add(elemento)`

- Rimuovere un elemento
  `I.remove(elemento)`

- Creare nuovi insiemi che siano unione, intersezione e differenza di `I` con un altro insieme `J`
  `I.union(J)`
  `I.intersection(J)`
  `I.difference(J)`

Insiemi

```
I = {2,3,4,5}
print(I)
I.add(3)
print(I)
I.remove(3)
print(I)
I.remove(2)
print(I)
J = {'a', 'b', 'c', 5, 2}
print("union:", I.union(J))
print("intersection:", I.intersection(J))
print("difference:", I.difference(J))
```

```
{2, 3, 4, 5}
{2, 3, 4, 5}
{2, 4, 5}
{4, 5}
union: {2, 4, 5, 'a', 'c', 'b'}
intersection: {5}
difference: {4}
```

Dizionari e insiemi

Sia su dizionari che su insiemi è definita la funzione `len` che restituisce il numero di elementi del dominio del dizionario o dell'insieme:

`len({1,2,3,4})` → `4`

`len({1:2,3:4})` → `2`

Il dominio di un dizionario è un insieme che si può costruire a partire dal dizionario con l'operatore `set`

`set({1:2,3:4,5:6})` → `{1,3,5}`

L'insieme vuoto si denota `set()`

Il dizionario vuoto si denota con `{}`

Si può convertire un insieme in una lista con `list()`

`list({1,2,3,4})` → `[1,2,3,4]`

Dizionari e insiemi

```
print(len({1,2,3,4}))
print(len({1:2,3:4}))
print(set({1:2,3:4,5:6}))
print({1,2,3}.intersection({4,5,6}))
print(list({1,2,3,4}))
```

```
4
2
{1, 3, 5}
set()
[1, 2, 3, 4]
```

File

Dal punto di vista di un programma Python un file consiste in una sequenza ordinata di valori.

Più precisamente, un file può essere:

- una sequenza di byte, codificati come numeri interi con valori $\in [0,255]$
- una sequenza di caratteri (file di testo)

File di testo

Per i file di testo avremo due tipi di operazioni principali:

- lettura: acquisizione di una **sequenza** di caratteri del file, sottoforma di una **stringa**
- scrittura: memorizzazione nel file di una sequenza di caratteri contenuti in una **stringa**

Accesso al file

La lettura o la scrittura di dati su un file avvengono in tre fasi:

1. apertura del file, per mezzo della funzione built-in `open`

2. esecuzione di una o più operazioni di lettura o scrittura, per mezzo delle opportune funzioni built-in

3. chiusura del file, per mezzo della funzione built-in `close`

`open`

La funzione `open` restituisce un valore strutturato contenente alcune informazioni sul file.

Sintassi:

```
variabile = open(nome_file, modalita)
```

- `variabile`: il nome della variabile che verrà associata al file
- `nome_file`: una stringa contenente il nome del *file*
- `modalita`: una stringa che indica la modalità di apertura (lettura o scrittura)

`open`

Il nome del file che si desidera aprire deve essere passato come argomento della funzione `open` sotto forma di stringa.

Il nome del file può essere:

- assoluto, cioè preceduto dalla sequenza (detta anche path) dei nomi delle directory che lo contengono a partire dalla directory radice del file system, scritta secondo la sintassi prevista dal sistema operativo del proprio calcolatore
- relativo, cioè composto dal solo nome del file: questo è possibile solo se la funzione `open` è chiamata da un programma o da una funzione che si trovi nella stessa directory che contiene il file da aprire

open

Nel caso di un file di nome `dati.txt` che si trovi nella directory `C:\Users\Erika\` di un sistema operativo Windows:

- il nome relativo è `dati.txt`
- il nome assoluto è `C:\Users\Erika\dati.txt`

Se un file con lo stesso nome (`dati.txt`) è memorizzato nella directory `/users/Erika/` di un sistema operativo Linux oppure Mac OS:

- il nome relativo è ancora `dati.txt`
- il nome assoluto è `/users/Erika/dati.txt`

open

È possibile aprire in modalità di lettura solo un file esistente. Se il file non esiste si otterrà un errore.

La modalità di scrittura consente invece anche la creazione di un nuovo file. Più precisamente, attraverso la modalità di scrittura è possibile:

- creare un nuovo file
- aggiungere dati in coda a un file già esistente
- sovrascrivere (cancellare e sostituire) il contenuto di un file già esistente

`open`

Nella chiamata di `open` la modalità di accesso è indicata (come secondo argomento) da una stringa composta da un singolo carattere:

- "r" (read): lettura (se il file non esiste si ottiene un errore)
- "w" (write): (sovra) scrittura
  - se il file non esiste viene creato
  - se il file esiste viene sovrascritto, cancellando i dati contenuti in esso
- "a" (append): scrittura (aggiunta)
  - se il file non esiste viene creato
  - se il file esiste i nuovi dati saranno aggiunti in coda a quelli già esistenti

open

```
f = open("dati.txt","w")
print(f)
!ls
```

```
<_io.TextIOWrapper name='dati.txt' mode='w' encoding='UTF-8'>
dati.txt  sample_data
```

open

```
f = open("dati.txt","w")
print(f)
!ls
%cd sample_data
!ls
f2 = open("README.md","r")
print(f2)
```

```
<_io.TextIOWrapper name='dati.txt' mode='w' encoding='UTF-8'>
dati.txt  sample_data
/content/sample_data
anscombe.json                   dati.txt                README.md
california_housing_test.csv     mnist_test.csv
california_housing_train.csv    mnist_train_small.csv
<_io.TextIOWrapper name='README.md' mode='r' encoding='UTF-8'>
```

open

```
f = open("dati.txt","w")
print(f)
!ls
%cd sample_data
!ls
f2 = open("README.md","r")
print(f2)
f3 = open("README.MD","r")
print(f3)
```

```
<_io.TextIOWrapper name='dati.txt' mode='w' encoding='UTF-8'>
anscombe.json                   dati.txt                README.md
california_housing_test.csv     mnist_test.csv
california_housing_train.csv    mnist_train_small.csv
[Errno 2] No such file or directory: 'sample_data'
/content/sample_data
anscombe.json                   dati.txt                README.md
california_housing_test.csv     mnist_test.csv
california_housing_train.csv    mnist_train_small.csv
<_io.TextIOWrapper name='README.md' mode='r' encoding='UTF-8'>
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-6-851e1e950b4a> in <module>()
      6 f2 = open("README.md","r")
      7 print(f2)
----> 8 f3 = open("README.MD","r")
      9 print(f3)

FileNotFoundError: [Errno 2] No such file or directory: 'README.MD'

SEARCH STACK OVERFLOW
```

close

Quando le operazioni di lettura o scrittura su un file sono terminate, il file deve essere chiuso attraverso la funzione built-in `close`. Questo impedirà l'esecuzione di ulteriori operazioni su tale file, fino a che esso non venga eventualmente riaperto.

Sintassi:

```
variabile.close()
```

dove `variabile` deve essere la variabile usata nell'apertura dello stesso file attraverso la funzione `open`

Esempio

```
f = open ("dati.txt", "r")
#operazioni di I/O su dati.txt
f.close()
```

close

```
!ls
f = open("README.md","r")
print(f)
f.close()
```

```
anscombe.json                   dati.txt                    README.md
california_housing_test.csv     mnist_test.csv
california_housing_train.csv    mnist_train_small.csv
<_io.TextIOWrapper name='README.md' mode='r' encoding='UTF-8'>
```

`write`

In un file di testo che sia stato aperto in scrittura (in modalità "w" oppure "a") è possibile scrivere dati sotto forma di stringhe (cioè sequenze di caratteri) attraverso la funzione built-in `write`

Sintassi:

```
variabile.write(stringa)
```

- `variabile` è la variabile associata al file
- `stringa` è una stringa contenente la sequenza di caratteri da scrivere nel file

La chiamata di `write` con un file aperto in lettura (in modalità "r") produce un messaggio di errore.

write

```
!ls
f = open("dati.txt","w")
f.write("dati dati dati")
f.close()
!cat dati.txt
```

```
dati.txt  sample_data
dati dati dati
```

write

```
!ls
f = open("dati.txt","w")
f.write("dati dati dati")
f.close()
!cat dati.txt
f2 = open("dati.txt","r")
f2.write("ancora dati dati dati")
f2.close()
```

```
dati.txt  sample_data
dati dati dati
---------------------------------------------------------------------------
UnsupportedOperation                      Traceback (most recent call last)
<ipython-input-8-2c6360666dab> in <module>()
      5 get_ipython().system('cat dati.txt')
      6 f2 = open("dati.txt","r")
----> 7 f2.write("ancora dati dati dati")
      8 f2.close()

UnsupportedOperation: not writable
```

SEARCH STACK OVERFLOW

write

```
!ls
f = open("dati.txt","w")
f.write("dati dati dati")
f.close()
!cat dati.txt
f2 = open("dati.txt","a")
f2.write("ancora dati dati dati")
f2.close()
!cat dati.txt
```

```
dati.txt  sample_data
dati dati datidati dati datiancora dati dati dati
```

Lettura da file

Le operazioni di lettura da file possono essere eseguite attraverso tre diverse funzioni built-in:

- `read`
- `readline`
- `readlines`

Queste tre funzioni restituiscono una parte o l'intero contenuto del file sottoforma di una stringa oppure di una lista di stringhe.

Il valore restituito dalle funzioni di lettura viene di norma memorizzato in una variabile per poter essere successivamente elaborato.

`read`

La funzione `read` acquisisce l'intero contenuto di un file, che viene restituito sotto forma di una stringa.
Le eventuali interruzioni di riga presenti nel file vengono codificate con il carattere newline "\n".

Sintassi:
`variabile.read()`
dove `variabile` è la variabile associata al file.

read

```
!ls
f_w = open("dati.txt","w")
f_w.write("dati dati dati")
f_w.write("\n")
f_w.write("ancora dati dati dati\n")
f_w.close()
!cat dati.txt
!echo "-----"
f_r = open("dati.txt","r")
file_content = f_r.read()
print(file_content)
f_r.close()
```

```
dati.txt  sample_data
dati dati dati
ancora dati dati dati
-----
dati dati dati
ancora dati dati dati
```

`readline`

La funzione `readline` acquisisce una singola riga di un file, restituendola sotto forma di una stringa.

Per “riga” di un file si intende una sequenza di caratteri fino alla prima interruzione di riga, oppure, se il file non contiene interruzioni di riga, l’intera sequenza di caratteri contenuta in esso.

Nel primo caso anche l’interruzione di riga, codificata con il carattere newline, farà parte della stringa restituita da readline.

Sintassi:
`variabile.readline()`
dove `variabile` indica come al solito la variabile associata al file.

readline

```
!cat dati.txt
!echo '----'
f_r = open("dati.txt","r")
line = f_r.readline()
print(line)
line = f_r.readline()
print(line)
f_r.close()
```

```
dati dati dati
ancora dati dati dati
----
dati dati dati

ancora dati dati dati
```

`readlines`

La funzione `readlines` acquisisce l'intera sequenza di caratteri contenuta in un file, suddivisa per righe, e la restituisce sotto forma una lista di stringhe, ciascuna delle quali contiene una riga del file (incluso il carattere newline).

Sintassi:
`variabile.readlines()`
dove `variabile` è la variabile associata al file.

La funzione `readlines` è utile quando si desidera elaborare separatamente ogni riga di un dato file.
A differenza di `read` consente l'acquisizione dell'intero file con una sola chiamata.

readlines

```
!cat dati.txt
!echo '----'
f_r = open("dati.txt","r")
lines = f_r.readlines()
print(lines)
print(lines[1])
f_r.close()
```

```
dati dati dati
ancora dati dati dati
----
['dati dati dati\n', 'ancora dati dati dati\n']
ancora dati dati dati
```

Eccezioni

Alcune possibili eccezioni per i programmi in Python sono:

- la divisione di un valore per zero, come:
  ```
  print(55/0)
  ```
  che produce
  ```
  ZeroDivisionError: integer division or modulo
  ```
- la richiesta di un elemento di una lista con un indice errato, come:
  ```
  a = [3,2]
  print(a[5])
  ```
  che produce
  ```
  IndexError: list index out of range
  ```
- la richiesta di una chiave non esistente in un dizionario:
  ```
  b = {'nome':'Antonio'}
  print(b['eta'])
  ```
  che produce
  ```
  KeyError: 'eta'
  ```

FileNotFoundError

```
nome_file = input("Inserire il nome per il file da leggere: ")
try:
    f = open(nome_file, 'r')
except FileNotFoundError:
    print("Il file non esiste!")
```

```
Inserire il nome per il file da leggere: numeri.txt
Il file non esiste!
```

with

L'istruzione `with` è una struttura di controllo del flusso di esecuzione

Sintassi:

```
with espressione [as variabile]:
    blocco_di_istruzioni
```

dove `espressione` è una espressione che deve produrre un oggetto che supporta il *context management protocol* cioè un oggetto che abbia i metodi `__enter__()` e `__exit__()`

Il valore di ritorno di `__enter__()` viene assegnato a `variabile` (se fornita).

L'istruzione `with` garantisce che se il metodo `__enter__()` termina senza errori allora la funzione `__exit__()` sarà sempre invocata.

with

```
with open('dati.txt', 'r') as f:
    for line in f:
        print(line)
```

```
dati dati dati

ancora dati dati dati
```

Dopo l'esecuzione del `with` l'oggetto file `f` verrà automaticamente chiuso, anche se dovesse verificarsi una eccezione nel ciclo `for`

with

```
with open('numeri.txt', 'r') as f:
    for line in f:
        print(line)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-31-ba50b116c54b> in <module>()
----> 1 with open('numeri.txt', 'r') as f:
      2     for line in f:
      3         print(line)

FileNotFoundError: [Errno 2] No such file or directory: 'numeri.txt'
```

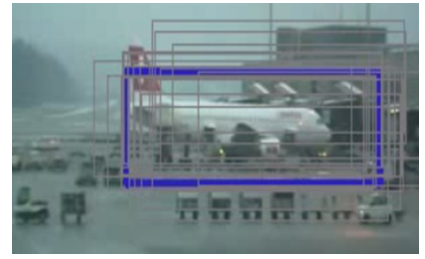
SEARCH STACK OVERFLOW



UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

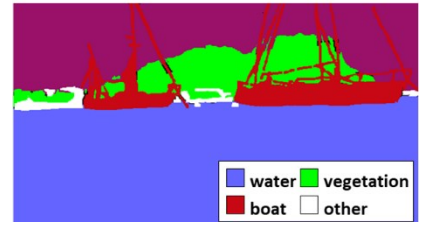
Corso di Sistemi Informativi
A.A. 2018/19

Docente
Domenico Daniele Bloisi


Dizionari e File in Python

Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)

Marzo 2019