ZerotoHero_2

February 19, 2020

0.1 Virtual Enviroment

[ ]:

[ ]:

1 installare Anaconda

- conda env list per visualizzare elenco enviroment presenti sulla macchina
- conda create –name Corso python=2.7
- source activate Corso / source deactivate (dall'ìnterno dell'enviroment) conda prevede conda activate e source deactivate

Conda è un secondo repository rispetto a *pip* ergo alcuni pacchetti possono avere un ritardo oppure non essere presenti nel database. In questo caso, dopo aver attivato la *virtualenv corretta* e poi usare **pip**.

```
import  utility.contaparole as conta
import utility.calcolatrice as calcolatrice
```

...

Nella directory ci sarà un file **contaparole.py** e **calcolatrice.py**. L'utilizzo di *as* risulta utile quando hanno lo stesso nome.

2 Package

Se voglio avere una cartella in cui mettere tutte le mie utility basta che al suo interno aggiunga un file chiamato **___init___.py** anche se vuoto; questo mi permette, come nell'esempio precedente dove ho una cartella utility che contiene i miei moduli, di richiamarli nella forma **utility.contaparole**.

Se non voglio rendere visibili tutti i sorgenti presenti nella cartella scriverò, nel file init che abbiamo visto prima la stringa (dunder all …)

Solo i sorgenti elencati potranno essere importati e visti!

~~~ **all** = [calcolatrice,]

Quando *importiamo* un modulo python esegue tutto il codice contenuto; ergo se nel modulo sono presenti dei print vengono "eseguiti" nella fase di importazione!
~~~

~~~ print(“nome modulo: %s” %(**name**))

~~~ if **name** = “**main**”: calcola()

Se lancio direttamente il modulo partirà automaticamente la calcolatrice, diversamente entrerò nel loop di scelta while!

~~~ def calcola(): #chiedo il primo numero primo = int(input(“immetti un numero:”)

```
#chiedo il secondo numero
secondo = int(input("immetti un altro numero: ")

operazione == 0

while(operazione == 0):
    operazione = input("che operazione desideri eseguire\n(1. somma...)

    if(operazione == 1):
        risultato = primo + secondo
    elif(operazione == 2):
        risultato = primo - secondo...
    else:
        operazione = 0
        #messaggio di errore
        print("\nSelezionare un'operazione valida\n")
print("Il risultato è: %d" % (risultato))
```

[ ]:

~~~ def conta(): frase = input(“inserire una frase:”)

```
parole = frase.split(" ")
conteggio = len(parole)

print("nella frase %s ci sono %d parole." % (frase, conteggio))

numero = 1

for parola in parole:
    print("%d %s " % (numero, parola))
    numero += 1
```

[6]:
```
voti = {"luca": 7, "andrea": 9, "silvia": 7, "luigi": 5}
promossi = {chiave:valore for chiave,valore in voti.items() if valore >= 6}
print(promossi)
```

```
{'luca': 7, 'andrea': 9, 'silvia': 7}
```

[5]:
```
elenco = [7, 4, 12, 45, 55]
validi = [x for x in elenco if x > 30]
print(validi)
```

```
[45, 55]
```

[20]:
```
for numero, valore in enumerate(elenco, start=10):
    print(numero, valore)
```

```
10 7
11 4
12 12
13 45
14 55
```

[29]:
```
persone = ["mario", "andrea", "franco", "mario", "andrea", "andrea"]
casa = set(x for x in persone)
```

[30]:
```
print(casa)
```

```
{'franco', 'mario', 'andrea'}
```

2.1 files

[5]:
```
%%writefile persone.txt
andrea
mario
anna
franco
mario
andrea
giovanni
```

```
Overwriting persone.txt
```

[6]:
```
cat persone.txt
```

```
andrea
mario
anna
franco
mario
andrea
giovanni
```

[20]:
```
univoci = set()

with open("persone.txt") as file_testo:
    for numero_riga, riga in enumerate(file_testo, 1):
        if riga in univoci:
            print("trovato duplicato alla riga", numero_riga)
        else:
```

```
            univoci.add(riga.strip())

print([x for x in univoci])
```

```
['giovanni', 'franco', 'mario', 'anna', 'andrea']
```

2.2 Parametri opzionali e valori di default

Vanno messi *dopo* i parametri obbligatori

[26]:
```
def sommatoria(primo, secondo=0, terzo=0):
    return primo + secondo + terzo
```

[27]:
```
print(sommatoria(3, terzo=5)) # salto il secondo parametro, devo dargli un␣
↪default per evitare ERROR!
```

```
8
```

[31]:
```
def sommatoriaINF(*values):
    somma = 0
    for n in values:
        somma += n
    return somma
```

[33]:
```
print(sommatoriaINF(2,3,5,12))
print(sommatoriaINF(1,2))
```

```
22
3
```

[34]:
```
def sommatoria2(*values):
    return sum(values)
```

[35]:
```
print(sommatoria2(1, 3, 5, 7))
```

```
16
```

[41]:
```
def popola(**valori): # asterisco singolo DEVE precedere asterisco DOPPIO
    for k, v in valori.items():
        print(k, "=", v)
    print(valori)
```

[42]:
```
popola(primo=1, secondo=2, terzo=5)
```

```
primo = 1
secondo = 2
terzo = 5
{'primo': 1, 'secondo': 2, 'terzo': 5}
```

3 DECORATORI

Sono una stringa che modifica il comportamento di classi, metodi, etc.

[43]:
```
def funzione(a,b):
    return a * b
```

[45]:
```
# se metto le parentesi chiamo la funzione, diversamente la uso come OGGETTO!
pippo = funzione
pluto = funzione
print(pippo(3,5))
```

```
15
```

I decoratori usano un oggetto come parametro di un altro oggetto!

3.1 Esempio di *CALLBACK*

[46]:
```
def chiamatore(a, b, operazione):
    return operazione(a, b)

def somma(a, b):
    return a + b

def moltiplicazione(a, b):
    return a * b
```

[48]:
```
print(chiamatore(5, 3, somma))
```

```
8
```

[53]:
```
def scrittore(function):
    print("prima")
    function()
    print("dopo")


def chiamata():
    print("durante")
```

[1]:
```
def scrittore(function):
    def function_wrapper():
        print("prima")
        function()
        print("dopo")
    return function_wrapper

@scrittore # prima esegue scrittore passandogli come parametro chiamata
```

```
def chiamata():
    print("durante")
```

[2]:
```
chiamata()
```

```
prima
durante
dopo
```