**6 30/11/2020 **

**ripetere task (loops)**

FOR (conosciamo esattamente quante volte ripetere)

per ripetere N volte tutte le istruzioni indentate sotto il for. Per la parola “Hello world”

word = input("What word to spell? ")

N=len(word)

for counter in range(N):

print(counter+1, " :")

print(" ", word[counter])

Range(N) significa incremento un ciclo che va da 0 a N-1 con step 1, ma posso personalizzare con

for counter in range(start, end, step):

*per avere un ciclo che va da start a end-1 con step pari a step*

*differenza tra equivalenza (o diversità) ed assegnazione* a==b a=b a!=b

non solo con contatore numerico: per quanti caratteri ha una stringa

for ch in "Hello world":

if ch != " ":

print(ch)

*ok anche for elemento in [423, 36, 8, 297, 1] per navigare elementi in una lista*

WHILE (ripetiamo in base al verificarsi di condizioni)

Evitando l’infinite loop / la variabile da testare nel ciclo (a differenza del contatore di FOR) va inizializzata

text = input("Proseguo? SI o NO: ")

while text == "SI":

print("ho fatto un giro")

text = input("Ancora? SI o NO: ")

text2 = input("Puoi ancora interrompere… interrompo? SI o NO: ")

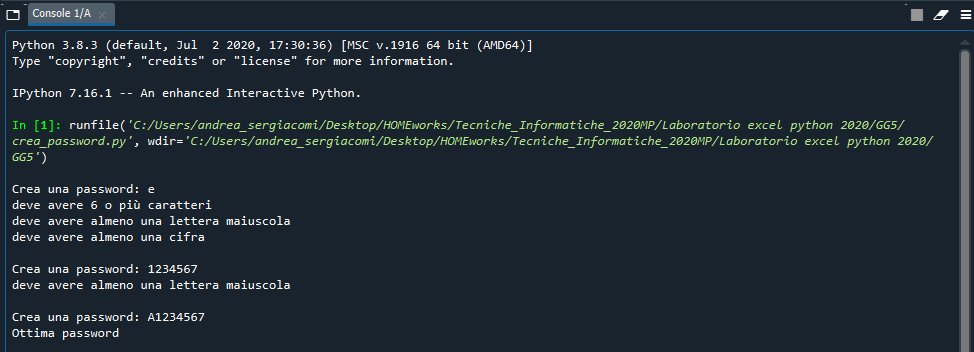
if text2 == "SI":

break

**break** interrompe il ciclo. Un’altra parola riservata **continue** invece forza il ciclo a proseguire



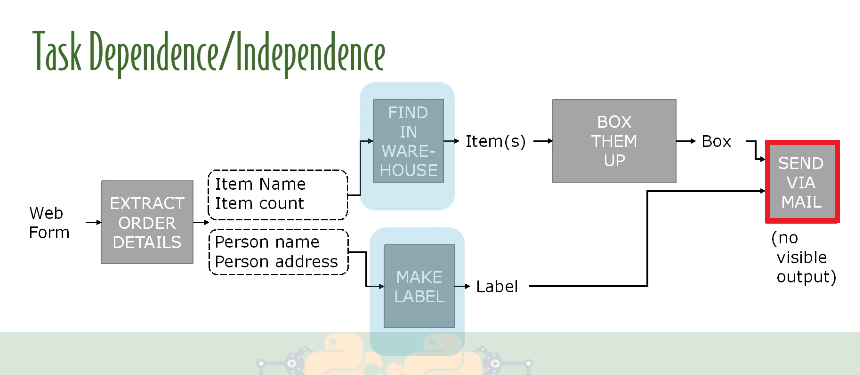
Crea\_password.py



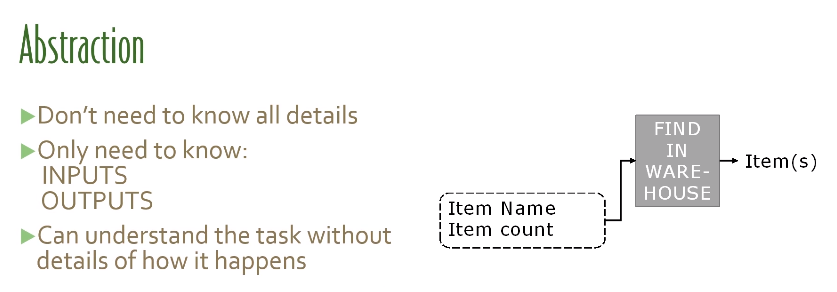
**Organizzare blocchi riusabili (modules of code): le funzioni**

PROCESSO suddiviso in tasks (es. acquistare online)

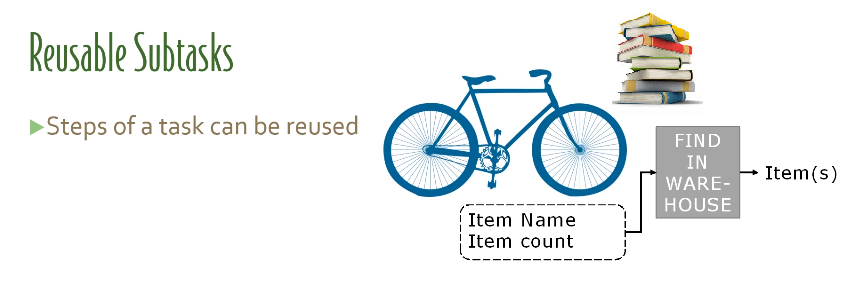
1 parallelizzazione o dipendenza dei tasks



2 astrazione del task



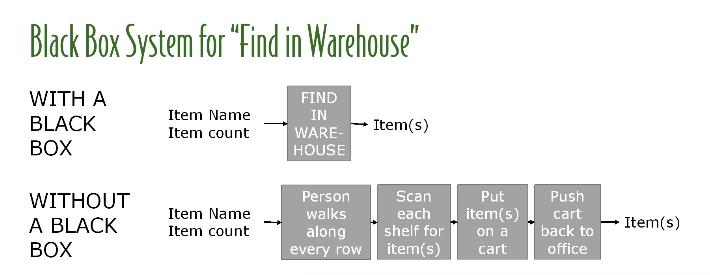
3 tasks riusabili



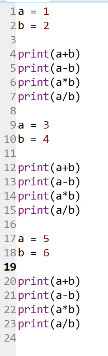
Il task (compito) diventa una Black Box (scatola nera) descritta solo da:

* input (parametri: variabili che assumono un valore solo dopo una chiamata alla funzione, se presenti)
* output (risultato: il valore ritornato, in alcuni casi anche **None**)
* operazioni & calcoli (nome evocativo della funzione richiamata)

Non serve addentrarsi nei dettagli (semmai dentro la funzione faccio un breve commento per riassumere/spiegare i vari passaggi e tasks effettuati)



Un listato iterativo senza funzioni, ripetendo codice



La stessa cosa richiamando funzioni (e scrivendo il codice basilare solo una volta):

# ESEMPIO FUNZIONI 4 OPERAZIONI ARITMETICHE

def somma(num1, num2):

# ===

# presi due numeri in input li sommo e restituisco il risultato

# ===

return num1+num2

def prodotto(num1, num2):

# ===

# presi due numeri in input li moltiplico e restituisco il risultato

# ===

return num1\*num2

def differenza(num1, num2):

# ===

# presi due numeri in input sottraggo il secondo al primo e restituisco il risultato

# ===

return num1-num2

def quoziente(num1, num2):

# ===

# presi due numeri in input divido il primo (dividendo) per il secondo (divisore) e restituisco il risultato

# ===

return num1/num2

# NB ESISTONO ANCHE FUNZIONI SENZA PARAMETRI NE’ RETURN

def input\_numeri():

onenum = float(input("Dammi il primo numero: "))

twonum = float(input("Dammi il secondo numero: "))

print("Il risultato dell’addizione è:", somma(onenum, twonum))

print("Il risultato della moltiplicazione è:", prodotto(onenum, twonum))

print("Il risultato della sottrazione è:", differenza(onenum, twonum))

print("Il risultato della divisione è:", quoziente(onenum, twonum))

print("Eseguirò le 4 operazioni aritmetiche su 2 numeri presi in input")

quante = int(input("Quante volte inserirai i 2 numeri? "))

for i in range(quante):

input\_numeri()

Le variabili valgono all’interno degli ambiti di definizione (corpo principale vs funzione):

# ESEMPIO AMBITO <SCOPE> DELLE VARIABILI

def fairy\_tale():

peter = 5

print(peter)

peter=30

fairy\_tale()

print(peter)

Una funzione può essere indentata dentro un’altra

*def funzione1():*

*def funzione2():*

*…*

*Ma poi per eseguire funzione2 devo chiamare funzione1 dal corpo principale*

Una funzione può anche essere passata come parametro

def panino(tipo\_di\_panino):

print("--------------------")

print(tipo\_di\_panino())

print("--------------------\n")

def cheeseburger():

my\_cb = "cheddar\nlattuga\ncipolla\nbacon\nketchup\nhamburger di scottona"

return my\_cb

def sandwitch\_tonno():

my\_st = "lattuga\nmayonese\ntonno"

return my\_st

print("Cheeseburger")

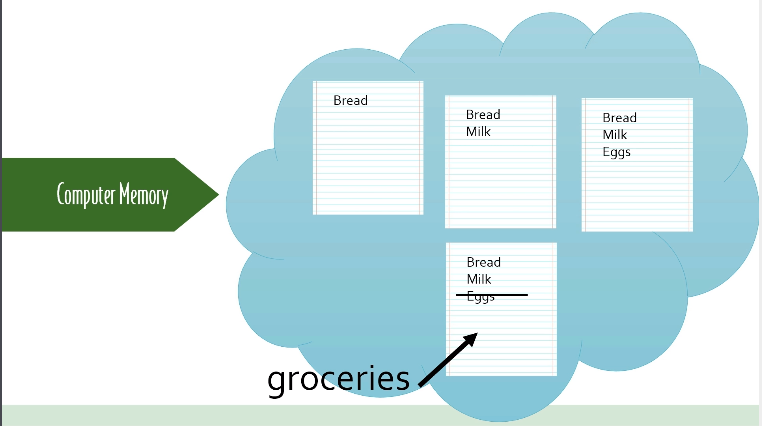
panino(cheeseburger)

print("Sandwitch al tonno")

panino(sandwitch\_tonno)

**IMMUTABLE & MUTABLE OBJECTS: introduzione a liste e dizionari**

Oggetti immutabili (Booleans, Integers, Floats, Strings, Tuples)



Ad ogni nuova istanza resta in memoria il vecchio valore (ma perde il legame con la variabile). Python si preoccupa di cancellare i vecchi oggetti, attraverso un processo chiamato garbage collection.

# esempio TUPLA (notate le parentesi tonde)

a\_tuple = ("milk", "eggs")

print(id(a\_tuple))

# 2159363028872 (ID è la rappresentazione numerica di un indirizzo di memoria di uno specifico oggetto)

a\_tuple = ("milk", "eggs", "bread")

print(id(a\_tuple))

# 2159363351248 (è cambiato)

a\_tuple = a\_tuple + ("bacon")

print(id(a\_tuple))

# 2159363456348 (è cambiato ancora!)

A che servono le tuple? Ad es. per memorizzare coordinate geografiche (21.407887, -111.111316) oppure gli indici di una parola in un libro, pagina, n° riga, n° colonna ("ISBN 0-7645-2641-3", 144, 10, 24)… cose che di solito non mutano.

Oggetti mutabili (Liste e Dizionari)

Una volta creato il valore dell’oggetto può comunque variare

# esempio LISTA (parentesi quadre)

a\_list = ["milk", "eggs"]

print(id(a\_list))

# 2159363498056

a\_list = ["milk", "eggs", "bread"]

print(id(a\_list))

# 2159363284680 (è cambiato)

a\_list.append("bacon")

print(id(a\_list))

# 2159363284680 (è lo stesso di prima!)

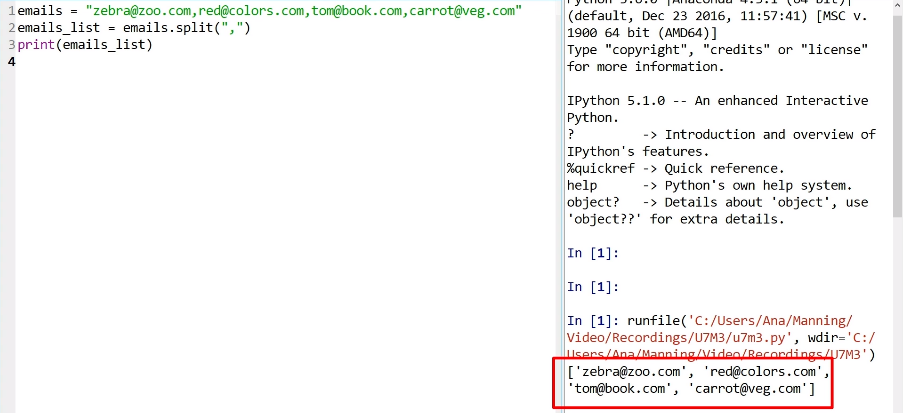
*NB == e is (!= e is not): a == b confronta i valori a is b confronta l’identità dell’oggetto, cioè se ha uguale id (entrambi restituisco true o false)*

Tutto questo vale per aggiungere, cancellare, riorganizzare ad esempio ordinare) ed è una gestione della memoria più efficiente rispetto a copiare di nuovo tutto in un nuovo oggetto

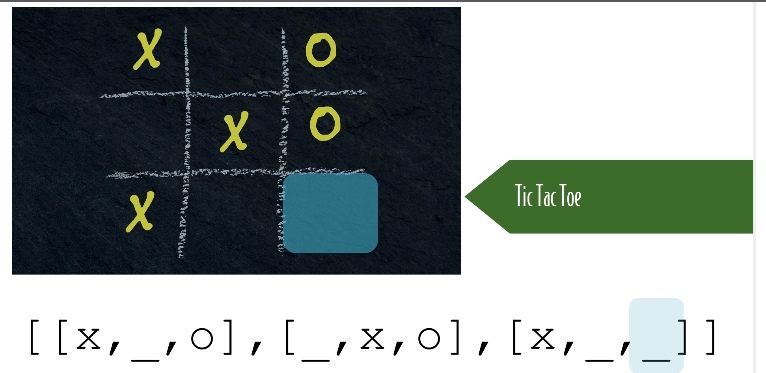
Valgono le stesse regole delle stringhe: posso usare LEN(a\_list) # 4 dopo l’aggiunta di bacon; a\_list[0] # vale milk – si parte da 0 che è il primo item.

Ma posso anche contare quante volte un elemento compare in una lista a\_list.count("bacon") # 1 volta; qual è l’indice in cui si trova la prima occorrenza di un elemento a\_list.index("milk") # ovvero 0; aggiungere item con append, inserire item a\_list.insert(1, "potatoes") # inserirebbe “potatoes” tra “milk” ed “eggs”, fondere due liste con extend a\_list.extend(lista\_spesaB) # supponendo che lista\_spesaB sia un’altra LIST; rimuovere item a\_list.pop() # l’ultimo della lista o a\_list.pop(1) # quello con indice 1; cambiare valori in una certa posizione a\_list[0]="croissant" # mette croissant al posto di milk; ordinare in ordine crescente gli elementi a\_list.sort(); invertire gli elementi a\_list.reverse();

Posso agevolmente trasformare stringhe con carattere separatore (es. CSV) in liste con il comando lista=stringa.split(",") # dove la virgola è il carattere separatore



Posso costruire LISTE di LISTE



Con i dizionari estendo il concetto di liste, mappando un item, la chiave, ed un altro item, il valore: ad esempio una lista della spesa con alimenti e quantità. Posso solo abbinare 1 oggetto CHIAVE ad 1 oggetto VALORE (ma l’oggetto valore potrebbe essere una TUPLA di più valori)

# creo un dizionario – prima vuoto – nota le parentesi grafe – non c’è un ordine se non quello di inserimento

grocery = {}

grocery = {"milk":1, "eggs":12, "bread":2}

grocery["bacon"]=44 # aggiunge in coda chiave e valore (o modifica il valore di una chiave se già esistente, perché il DIZIONARIO è mutabile, ma la KEY è unica e immutabile)

print(grocery) # {'milk': 1, 'eggs': 12, 'bread': 2, 'bacon': 44}

print(len(grocery)) # 4

removed=grocery.pop("bacon") # POP rimuove chiave e valore

print(removed) # 44 (ma può memorizzare il valore che aveva la chiave rimossa)

grocery["eggs"]= grocery["eggs"]-2 # sottrae 2 uova dal valore

print(grocery) # {'milk': 1, 'eggs': 10, 'bread': 2}

print(grocery.keys()) # restituisce solo le chiavi: dict\_keys(['milk', 'eggs', 'bread'])

# grocery.keys è una specie di lista

for one\_item in grocery.keys():

print(one\_item)

# ma posso crearmi una vera e propria lista di chiavi con un casting

all\_items = list(grocery.keys())

# analogamente c’è il comando VALUES

all\_items\_values = list(grocery.values())

print(all\_items\_values)

# frequency dictionary

lyrics = "Happy birthday to you Happy birthday to you Happy birthday dear friend Happy birthday to you"

counts = {}

words = lyrics.split(" ")

for w in words:

w=w.lower()

if w not in counts:

counts[w] = 1

else:

counts[w] += 1

print(counts)

# {'happy': 4, 'birthday': 4, 'to': 3, 'you': 3, 'dear': 1, 'friend': 1}

# ALIASING

# puntamento in memoria ad oggetti per variabili immutabili (numeri interi)

a=1

print("ID a")

print(id(a))

b=a #stessa cosa se pongo b=1

print("ID b - dopo b=a")

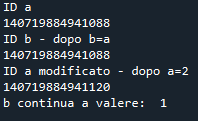
print(id(b))

a=2

print("ID a modificato - dopo a=2")

print(id(a))

print("b continua a valere: ", b)



# puntamento in memoria ad oggetti per variabili mutabili (lista)

genius = ["einstein","galileo"]

print("\nID GENIUS")

print(id(genius))

smart = genius

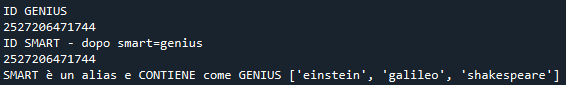
print("ID SMART - dopo smart=genius")

print(id(smart))

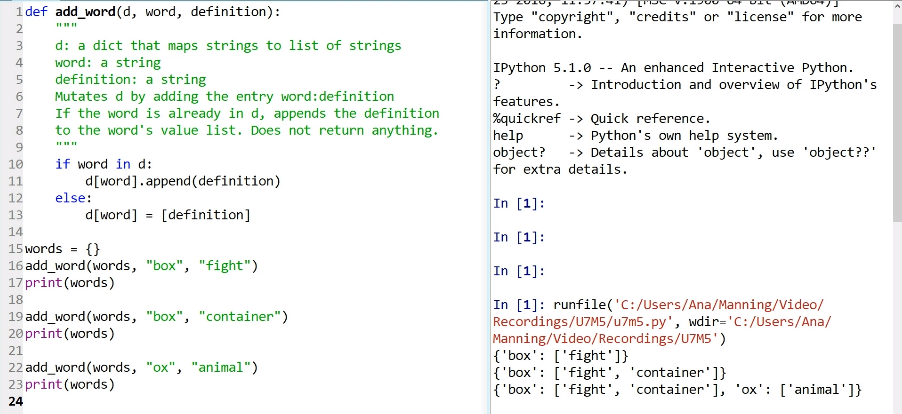
genius.append("shakespeare")

print("SMART è un alias e CONTIENE come GENIUS", smart)

# smart è un ALIAS di genius - mantiene stesso ID ovvero puntano allo stessa location in memoria



# ALIASING di oggetti mutevoli introduce cambiamenti anche se le variabili sono definite nel corpo principale e nella funzione (come parametro)



# per fare una copia di oggetti pesanti, come liste e dizionari, anziché usare aliasing

# o creo una nuova lista/dizionario contenente gli stessi elementi dell'altro oggetto (COPIA ESPLICITA)

personaggi = list(genius)

print("ID PERSONAGGI dopo copia esplicita lista")

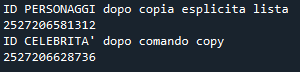
print(id(personaggi))

# o uso il comando COPY

celebrità = genius.copy()

print("ID CELEBRITA' dopo comando copy")

print(id(celebrità))



# SORTED - crea automaticamente una copia ma ordinata della lista

kid\_ages = [2, 1, 4]

sorted\_ages = sorted(kid\_ages)

print("original KID\_AGES", kid\_ages)

print("SORTED\_AGES", sorted\_ages) # [1, 2, 4]

# ATTENZIONE: esempio “rimuovere tutti gli elementi con valore 1 da dizionari o liste” -> i cicli for non funzionano sull’oggetto stesso

# DIZIONARIO SONGS = {"Wannabe":1, "Roar":1, "Let It Be":5, "Red Corvette":4}

for s in songs.keys():

if songs[s] == 1:

songs.pop(s)

# (ERRORE: Python non permette di cambiarne dimensione mentre si sta iterando sull’oggetto)

# LISTA SONGS = [1, 1, 5, 4]

for s in songs:

if s == 1:

songs.remove(s)

# (RISULTATO ERRATO [1, 5, 4]: alla seconda iterazione, dopo aver rimosso il primo 1, si è all’indice 1 che contiene 5: il secondo 1 non viene rimosso)

# LISTA SONGS = [1, 1, 5, 4]

songs\_copy = songs.copy()

songs = []

for s in songs\_copy:

if s != 1:

songs.append(s)

# (CORRETTO: procedo al contrario, ovvero copio in una nuova struttura, svuoto la lista originaria, ci appendo – a partire dai dati nella copia - solo gli item che rispettano i requisiti, cioè diversi da 1)