1

Programmazione in Python 3

Enrico Ferreguti

--

Programmazione in Python 3

Fondamenti, Installazione e ambiente operativo

Enrico Ferreguti

PYTHON

Python è un linguaggio di programmazione potente e divertente, che trova utilizzo sia nello sviluppo di applicazioni web che di software desktop. E' possibile ritrovarlo come interfaccia di programmazione di librerie tipo GDAL/OGR, JTS e GEOS. Le caratteristiche principali sono:

- chiarezza e leggibilità della sintassi
- è orientato alla programmazione ad oggetti ma anche alla programmazione funzionale
- gestione degli errori in fase di esecuzione
- tipi dato dinamici e di alto livello
- codice modulare e riusabile
- disponibilità sterminata di librerie immediatamente disponibili (https://pypi.python.org/pypi)
- usato come linguaggio di *scripting* in molti software

RISORSE

Libri

- Imparare Python, Roberto Allegra
- Dive into python3

--

Tutorial

- Il tutorial ufficiale di Python
- Pensare da informatico. Versione Python
- Dive into Python - Python per programmatori esperti
- Google Python Course

--

Riferimenti

- Cheatsheet - riferimento rapido python 3.5
- The Hitchhiker's Guide to Python

INSTALLAZIONE

Python è usato come linguaggio di scripting all'interno di altri software (per esempio ArcGis e QGIS per restare nell'ambito geospaziale) e di solito la prima esperienza di questo linguaggio è in relazione a questi programmi per realizzare personalizzazioni o processi automatizzati.
Il linguaggio però può essere installato in modo *standalone* scaricando l'interprete dal seguente indirizzo: https://www.python.org/downloads/

Mentre in Linux e MacOS Python è maggiormente integrato nel sistema operativo mentre in Windows è più facile far coesistere diverse versioni differenti, ma fondamentalmente l'esperienza è fondamentalmente la stessa ed eventuali differenze sottostanti sono gestite dall'interprete in modo da eliminare o quanto meno limitare al minimo le incompatibilità tra vari sistemi. (per esempio: os)

Si può affermare che un codice Python può funzionare con opportuni accorgimenti in tutti i sistemi operativi.

Controlliamo se tutto è a posto. apriamo una finestra di comando e digitiamo il comando python e dovrebbe comparire il tipico pronto della console del linguaggio : >>>

STRUMENTI

In Python l'indentazione delle istruzioni, oltre a rendere più leggibile il codice ha una precisa funzione sintattica. E' quindi necessario disporre di un'editor di testo per programmatori che faciliti l'indentazione evitando situazioni che protrebbero portare ad errori (per esempio mescolare tabulazioni e spazi), oltre a permettere l'evidenziazione della sintassi ed un minimo di introspezione del codice.

Noi utilizzeremo l'editor IDLE, realizzato completamente in Pyhton () e che presenta un'interfaccia semplice, una console integrata un debugger semplice studiato dalla comunità Python proprio per finalità educative e che viene installato insieme all'interprete del linguaggio

In alternativo possono essere scaricati in rete altri ambienti integrati di sviluppo (IDE) più complessi, adatti a sviluppare applicazioni professionali:

- GEANY - ATOM - BRACKETS - VISUAL STUDIO CODE

VERSIONI DEL LINGUAGGIO

Python nel corso della sua evoluzione è arrivato ormai release 3.8.2 della versione 3 e la versione 2 non è più supportata anche se ancora comunemente utilizzata (ArcGis 10)

Non supportato non significa che non funziona più ma che l'attività di bugfixing è cessata e di conseguenza eventuali vulnerabilità non saranno più corrette esponendo i sistemi a malfunzionamenti ed attacchi informatici

informazioni sulla migrazione

DOVE E' PYTHON

Contrariamente ai linguaggi compilati, che servono a produrre per mezzo di un compilatore un programma in linguaggio macchina, direttamente eseguibile da un microprocessore, Python è un linguaggio interpretato, ovvero che deve essere eseguito da un programma interprete.

I programmi in Python sono constituiti da file di testo con estensione .py che contengono le istruzioni da eseguire dall'interprete Python che è un eseguibile che valida il codice per evidenziare errori di sintassi, ovvero difformità dallo standard del linguaggio, ed appunto "interpreta" le istruzioni riga per riga svolgendo le funzionalità richieste con delle procedure interne all'interprete.

Un'altra modalità di funzionamento è quello da console, cioè digitando riga per riga i comandi da eseguire, una modalità utile per testare i comandi o gli ambienti in cui vengono eseguiti i programmi python

--

LA RIGA DI COMANDO

Ai fini del corso viene utilizzato l'interprete di python installato assieme a QGIS. Nei sistemi Linux e Macos, che hanno una migliore gestione dei pacchetti di installazione, python è installato come parte del sistema operativo, mentre nei sistemi Windows è installato come applicativo assieme a QGIS, come applicazione e non come parte del sistema operativo. Questo significa che potrebbero coesistere anche molte versioni di Python all'interno della macchina, causando la necessità di configurare opportunamente l'ambiente di esecuzione per evitare conflitti ed errori.

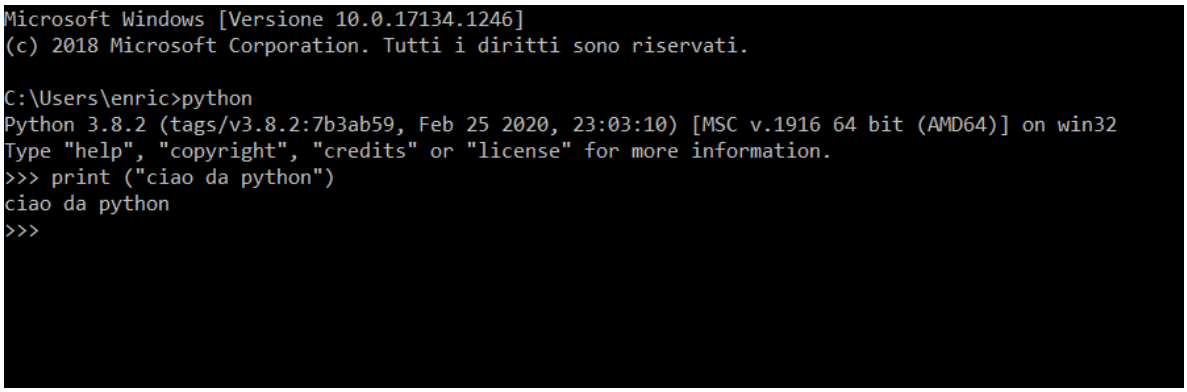
- In Linux e Macos, basta aprire il terminale e digitare `python` per entrare nell'interprete a riga di comando (oppure `python2` o `python3`)
- In windows bisogna tralasciare la console di sistema, ma andare sul menu di installazione di QGIS e cliccare su *OSGeo4W Shell* e digitare i comandi `py2_env.bat` o `py3_env.bat` a seconda che si desideri usare la versione 2 o 3 ed a questo punto digitare `python`

--

CIAO MONDO!

```
>>> print ("CIAO DA PYTHON!") <invio>
```

print è una funzione i cui parametri sono contenuti dalle parentesi ()
Nel nostro caso il parametro è unico ed è costituito da una stringa di testo da stampare sulla console. Dovremmo ottenere il seguente risultato



Il linguaggio PYTHON

- I tipi di dati
- le variabili

- i tipi strutturati
- il controllo del flusso di esecuzione
- funzioni, moduli e namespaces
- programmazione ad oggetti

THE ZEN OF PYTHON

```
>>> import this
```

I TIPI DI DATI

--

Numeri

```
10     #integer (numero intero)
25.45    #float (numero decimale)

# Per trasformare una stringa in numero
# o troncare un float
int("10")
int(34.354)

# L'aritmetica ha una notazione convenzionale
10 + 1
10 * 5
10*(45/3)+8*17

# Divisione tra interi in python 2
5 / 2 #2

# se uno dei numeri è float il risultato è float
10 / 3.0

# questo è equivalente a sopra
10 / float(3)

# in python 3 il risultato è implicito
```

--

Stringhe

```
#una stringa è una sequenza alfanumerica racchiusa tra apici '' o virgolette ""

# Concatenare le stringhe
"Ciao" + " " + "Mondo!"

# Formattazione di stringhe
"Ciao %s" % "Mondo!"

# Il testo è racchiuso da virgolette singole o doppie
```

```
'Python è "divertente"'

# Stringhe multi-linea
print ("""Questa stringa
è suddivisa in
varie righe""")

# sequenze di escape precedute da barra inversa "\"
print ("Questa stringa\nviene stampata\nin tre righe")
print ("Python è \"divertente\"")
print ("C:\\Users\\enrico")

# Si può usare una stringa grezza tralasciando le sequenze di escape
print (r"C:\Users\enrico")

#Uso del set di caratteri esteso (unicode - UTF8) (Python 2)
print (u"questo è qgis")
```

--

formattazione delle stringhe

il linguaggio dispone di funzioni molto potenti per assemblare tra loro le stringhe e formattare tipi non stringa (numeri, date)

```
a = 'Paolo'
b = "Verdi"
c = 172
d = 8.5676776
# concatenazione
print (u"il mio cognome è " + b)
print ("mi chiamo "+a+" "+b+" alto "+str(c)+" cm ed ho percorso "+str(d)+" km")
# metodo "vecchio"
print (u"il mio cognome è %s" % b)
print ("mi chiamo %s %s alto %d cm ed ho percorso %.1f km" % (a,b,c,d))
#metodo "nuovo"
print ("mi chiamo {1} {0} alto {2:d} cm ed ho percorso {3:.1f}
km".format(a,b,c,d))
```

--

TIPO *Boolean*

```
True

False #(None, [], {})

# operatori
not True # False
True and True # True
True and False # False
False and False # False
False or True # True
not (True or False) and True #False
```

```
1 == 1 #operatore di uguaglianza
1 != 1 #operatore di diversità
2 > 1  #maggiore
2 >= 1  #maggiore o uguale
1 < 2  #minore
1 <= 2  #minore o uguale
```

VARIABILI

--

Variabili

```
anno = "2017"

anno = 2017

print (anno)

anno_scorso = anno - 1

print (anno_scorso)
```

Le variabili in python hanno non sono "staticamente tipizzate", ovvero non devono essere dichiarate prima di usarle ne deve essere dichiarato il tipo. Sono dei contenitori che puntano a oggetti che potenzialmente possono essere di qualunque tipo. per conoscere il tipo dell'oggetto assegnato ad una variabile si usa type

```
type(anno)

<type 'int'>
```

I TIPI *STRUTTURATI*

--

Liste

una lista (list) e' un elenco ordinato di elementi, iterabile.

```
l = [3, 5, 4, 2, 1]
m = [[[0,3],[4,6],[5,7]],[[4,5],[6,8]]]

# Accesso agli elementi per posizione
# il primo elemento è 0, gli indici negativi partono dalla fine

l[0]       #3
l[0:3]     #[3, 5, 4]
l[-1]      #1
```

```
l[2:-2]    #[4, 2]
l[3:]      #[2, 1]
l[:-2]     #[3, 5, 4, 2]
m[0][2][0] #5
m[1][1]    #[6,8]
```

la stessa notazione è valida per ricavare sottostringhe

```
"abcdefghijklmnopqrstuvwxyz"[-5:] # "vwxyz"
```

--

Dizionari

un dizionario (dict) è un insieme strutturato di dati, iterabile.

```
d = {
    "chiave1": 10,
    "chiave2": 234.56,
    "chiave2": "contenuto",
    "chiave3": {
        "chiave3_1": "abcdef",
        "chiave3_2": "xyz"
    }
}

d["chiave2"]               # 234.56
d["chiave3"]["chiave3_2"]  # "xyz"
```

--

Manipolazione dei dati strutturati

```
l0 = [] # lista vuota
l1 = [1, 2]
l2 = [3, 5, 6]

d0 = {} # dizionario vuoto
d1 = {"nome":"giuseppe", "anni": 50}

l1 + l2          # [1, 2, 3, 5, 6]
l1 + 4 + l3      # TypeError
l1.append(10)    # [1, 2, 3, 5, 6, 10]
l1.append(l2)    # [1, 2, [3, 5, 6]]
3 in l2          # True
3 in l1          # False
len(l1 + l2)     # 5

d1["cognome"] = 'verdi'
d1["anni"] = 60 # {"nome":"giuseppe","anni":60,"cognome":"verdi"}
d1.keys()        # ["cognome", "anni", "nome"]
d1.values()      # ["verdi", 60, "giuseppe"]
len(d1)          # 3
```

Controllo del flusso

--

ESECUZIONE DI UN PROGRAMMA PYTHON

Le istruzioni python possono essere memorizzate in un file di testo con estensione .py ed essere eseguite in sequenza. L'esecuzione di un file .py può avvenire da riga di comando digitando

```
python file.py
```

Usando IDLE il programma può essere eseguito direttamente dall'editor ed il relativo l'output può essere esaminato sulla finestra di shell.

Le istruzioni python sono eseguite sequenzialmente una riga alla volta una dopo l'altra ma il flusso di esecuzione può essere modificato mediante le istruzioni IF / FOR / WHILE / TRY

--

IF THEN ELSE

```
latitude = 51.5
if latitude >= 0:
   zone_letter = 'N'
else:
   zone_letter = 'S'

print (zone_letter)
```

- l'indentazione è il sistema con cui Python raggruppa le istruzioni
- L'indentazione viene realizzata con spazi o tabulazioni
- ogni riga all'interno di un blocco base dev'essere indentata in ugual misura
- Usando un buon editor di testo per programmatori, copiare questo testo in un file vuoto, sa

--

Cicli

```
#for ELEMENTO in ITERABILE(LIST, DICT)

for carattere in "abcdefg":
    print (carattere)

for numero in [3, 5, 4, 2, 1]:
    print (numero)

#while CONDIZIONE_VERA(TRUE)

testo = "abcdefg"
while testo:
    testo = testo[:-1]
    print (testo)
```

un ciclo può essere interrotto con `break`

e si può saltare un'iterazione con `continue`

--

Eccezioni

Il debugging è una parte dell'attività di programmazione che mira ad individuare, interpretare e risolvere gli errori presenti codice da eseguire.

In python esistono due tipi di errori:

- errori di sintassi; che impediscono l'esecuzione di un programma
- errori *runtime* o eccezioni; che avvengono durante lo svolgimento del programma e che possono interrompere il flusso dello stesso o essere gestite per isolare il codice che potenzialmente potrebbe generarle ed adottare strategie alternative mirate.

--

Errore di Sintassi

```
>>> print( 0 / 0 ))
  File "<stdin>", line 1
    print( 0 / 0 ))
                  ^
SyntaxError: invalid syntax
```

Eccezione

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

--

Eccezioni comuni

NameError:

```
>>> print (c)
Traceback (most recent call last):
File "<pyshell#17>", line 1, in <module>
print (c)
NameError: name 'c' is not defined
```

TypeError:

```
>>> print(4+"a")
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    print(4+"a")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Eccezioni *Built-in*

--

Exception handling

In Pyhton le eccezioni possono essere gestite e manipolate mediante alcuni potenti comandi predefiniti:

- **raise** consente di creare una eccezione personalizzata e bloccare l'esecuzione del programma
- **assert** per bloccare in mdo condizionato l'esecuzione di un programma
- **try except else finally** per eseguire parti di codice in alternativa ad altre che determinano un eccezione

--

```
n = 2000
if n > 1000:
    raise Exception('n è {} ma non dovrebbe superare 1000'.format(n))

a = "abc"
assert(type(a) == int) , "a deve essere un numero"
print (a * 10)

try:
    print (100/0) #provoca ZeroDivisionError: integer division or modulo by zero
except:
    print ("il codice contenuto in try provoca un errore")

try:
    print (100/0)
except Exception as e:
    print ("il codice contenuto in try provoca l'errore" + str(e))
```

ESERCITAZIONI

--

TROVA I FATTORI DI UN NUMERO INTERO

```
num = 320

print("I fattori del numero",num,"sono:")
for i in range(1, x + 1):
    if x % i == 0:
        print(i)
```

--

TEST NUMERO PRIMO

```
num = 407

# I numeri primi sono più grandi di 1
if num > 1:
   # cerca il minimo comune divisore
   for i in range(2,num):
       if (num % i) == 0:
           print(num,"non è un numero primo")
           print(i,"volte",num//i,"è uguale a",num)
           break
   else:
       print(num,"è un numero primo")
else:
   print(num,"non è un numero primo")
```

--

CONTA LE VOCALI PRESENTI IN UNA STRINGA

```
stringa = "l'aida di giuseppe verdi"

# Inizializza un contatore
conta = 0
# Creating a set of vowels
vocali = "aeiouAEIOU"
# cicla attraverso tutti i caratteri di una stringa
for car in stringa:
    # Se il carattere è presente aumenta il contatore
    if car in vocali:
        conta = conta + 1

print("Numero di vocali:", conta)
```

--

TEST TROVA N VALORI MAGGIORI DI UNA LISTA

```
lista_iniziale = [2,4,66,7,8,12,89,3,4,1,90,45,8,34,5,6,77,84]
N = 4

lista_finale = []

for i in range (0, N):
  max1 = 0

  for j in range(len(lista_iniziale)):
    if lista_iniziale[j] > max1:
      max1 = lista_iniziale[j];

  lista_iniziale.remove(max1);
  lista_finale.append(max1)

print(lista_finale)
```

--

ORDINAMENTO *BUBBLE SORT*

```
lista = [3,5,2,7,44,6,44,3,46,4,67,23,67,5,3,9,56,23,67,0,1,76,4]

n = len(lista)

# Attraversa tutti gli elementi della lista

for i in range(n):

    # Gli ultimi i elementi
    for j in range(0, n-i-1):

        # attraversa l'array da 0 a n-i-1
        # scambia gli elementi a coppie quando
        # il primo elemento è più grande di quello dopo
        if lista[j] > lista[j+1] :
            lista[j], lista[j+1] = lista[j+1], lista[j]

print (lista)
```

Funzioni, moduli e namespaces

--

Funzioni esistenti (o predefinite)

le funzioni in python sono delle azioni di trattamento dei dati definite da un nome e degli argomenti da trattare separati da virgole e racchiusi tra parentesi. Le funzioni possono o meno restituire un valore di risposta.

```
# Funzioni esistenti (o predefinite)
print("stampa il numero",4)
sorted([3, 5, 4, 2, 1])
len(l1 + l2)
type(a)
globals()
```

--

Definizione della propria funzione

```
def ZonaUtm(longitudine, latitudine):
    numero_zona = ((longitudine + 180) / 6) % 60 + 1
    if latitudine >= 0:
        lettera_zona = 'N'
    else:
        lettera_zona = 'S'
    return '%d%s' % (int(numero_zona), lettera_zona)
```

Uso della funzione

```
ZonaUtm(11.55,45)   # 32
```

--

Namespaces

ogni oggetto in python *vive* all'interno di un blocco predeterminato di istruzioni, chiamato *namespace*e che idealmente corrisponde al livello di indentazione nel quale viene creato l'oggetto riferito agli oggetti *def* e *class*

All'interno di un programma possono coesistere numerosi namespaces separati tra loro in cui i nomi degli oggetti non sono in conflitto. Esiste almeno un namespace globale che contiene gli oggetti richiamabili da ogni parte del programma

```
def funzione1():
    var = 100
    print (var)

var = 50
print (var)
funzione1()
```

output:

```
50
100
```

```
globals() #simboli accessibili globalmente
locals() #simboli privati accessibili localmente (all'interno di una funzione)
```

--

test sui namespaces

```
a = 0
b = 1
c = 2

def stampa (a, b, c):
    print ('a:',a,'b:',b,'c:',c)

def test1():
```

```
    def test2(a,b,c):
        a += 1
        b += 1
        c += 1
        stampa(a, b, c)

    a = 6
    b = 5
    stampa(a, b, c)
    test2(a, b, c)

test1()
stampa(a, b, c)
```

--

Moduli

I moduli sono collezioni strutturate ed organizzate di codice python le cui definizioni possono essere importate all'interno di un programma

```
# Importa un modulo con chiamata ad una funzione contenuta
import math
math.floor(10.6)

# Importa singoli elementi da un modulo
from os import path
path.join('C:', 'Utenti', 'paolo', 'documenti')
'C:/Utenti/paolo/documenti'
```

l'organizzazione modulare è una delle caratteristiche del linguaggio. I moduli possono essere:

- predefiniti, già compresi nella dotazione di base del linguaggio
- esterni, contenuti nei path di sistema di Python (PATH, PYTHONPATH). Possono essere preimportati o importati da internet tramite pip/setuptools
- definiti localmente dall'utente in altri files python

Decoratori

In Python le funzioni sono considerate esse stesse degli oggetti infatti:

```
>>> math.sin
<built-in function sin>
```

e questo significa che:

- Possono essere passate come argomenti ad altre funzioni
- Possono essere definite all'interno di altre funzioni ed essere restituite come output di funzione

I Decoratori sono strumenti molto utili in Python che permettono ai programmatori di modificare il comportamento di una funzione o di una classe. I decoratori consentono di incapsulare una funzione dentro l'altra modificando il comportamento della funzione incapsulata senza modificarla permanentemente.

--

Sintassi dei decoratori

Nel codice seguente mio_decorator è una funzione *callable* ovvero chiamabile, invocabile, che aggiungerà l'esecuzione di qualche codice all'etto dell'esecuzione della funzione XYZ

```
@mio_decorator
    def XYZ():
    print ( "XYZ" )
```

ed equivale alla seguente funzione

```
def XYZ():
    print("XYZ")

hello_decorator = mio_decorator(hello_decorator)
```

--

Per esempio con un decoratore si può calcolare il tempo di esecuzione di una funzione:

Attenzione il nome riservato `args` restituisce la lista degli argomenti passati a quella funzione e `*` trasforma una lista in una sequenza di argomenti mentre il nome riservato `kwargs` restituisce la lista degli argomenti opzionali (*key-worded arguments*)passati a quella funzione e `**` trasforma una lista in una sequenza di argomenti opzionali. e la costruzione `*args, **kwargs` significa tutti gli argomenti che una funzione potrebbe avere

```
# importing libraries
import time
import math

def calcola_tempo_esecuzione(func):

    def funzione_incapsulante(*args, **kwargs):

        begin = time.time()
        valore_ottenuto = func(*args, **kwargs) #se la funzione ritorna un
valore si conserva per restuirlo poi
        end = time.time()
        print("Tempo totale di esecuzione della funzione", func.__name__, end -
begin)
    return valore_ottenuto

    return funzione_incapsulante


@calcola_tempo_esecuzione
def fattoriale(num):

    # imponiamo una pausa di due secondi per poter rendere evidente il tempo
```

```
    time.sleep(2)
    print(math.factorial(num))

fattoriale(10)
```

Classi ed istanze

- Classe
  - tipo di dato composto definito dall'utente in metodi ad attributi.
- Istanziare
  - creare un'oggetto di una determinata classe.
- Istanza
  - un'oggetto creato che appartiene ad una classe.
- Membri di una classe
  - Metodi
    - funzioni che costituisce un comportamento dell'istanza
  - Attributi
    - valori attribuiti ad un'istanza.
- Costruttore
  - metodo usato per definire nuovi oggetti.

--

definizione di classe e membri

```
class rettangolo:

   l = 0
   h = 0

   def __init__(self, l, h):
      self.l = l
      self.h = h

   def area(self):
      return self.l*self.h

   def perimetro(self):
      return(self.l+self.h)*2

   def scala(self, scala):
      self.l = self.l*scala
      self.h = self.h*scala
```

--

instanziare una classe

Una volta definita una classe è possibile .....

```
r = rettangolo(10,5) #istanziazione di un oggetto rettangolo
r.area()             #50
r.scala(2)
r.perimetro()        #60
r.l                  #20
r.h                  #10
```

--

Ereditarietà

è la capacità di definire una nuova classe come versione modificata di una classe già esistente

```
class stanza:

    def __init__(self, lung, larg):
        self.lung = lung
        self.larg = larg
        self.nome = "stanza"
    def nome(self):
        return self.nome
    def area(self):
        return self.lung * self.larg

class cucina(stanza):

    def __init__(self, lung, larg):
        super().__init__(lung, larg)
        self.nome = "cucina"

class camera(stanza):

    def __init__(self, lung, larg, abitante):
        super().__init__(lung, larg)
        self.nome = "bagno"
        self.abitante = abitante

class edificio:

    def __init__(self, stanze):
        self.stanze = stanze

    def area_tot(self):
        area = 0
        for stanza in self.stanze:
            area += stanza.area()
        return area

    def abitanti(self):
        ab = 0
        for stanza in stanze:
            if hasattr(stanza,'abitante'):
                ab += 1
        return ab
```

--

ispezione degli oggetti

individuazione del tipo di oggetto

```
type(oggetto)
```

lista dei membri di un' oggetto

```
dir(oggetto)
```

restituzione di un attributo di un oggetto con il suo nome

```
getattr(oggetto,'membro')
```

test se un attributo è presente nell'oggetto

```
hasattr(oggetto,'membro')
```

Esercitazione 1

calcolo delle abbreviazioni del nome e del cognome del codice fiscale italiano:

- scomposizione di una stringa in vocali e consonanti
- uso delle consonanti in successione, e se non sufficienti uso delle vocali in successione

--

```
# -*- coding: utf-8 -*-
'''
CORSO DI GEOPROCESSING - MASTER IN GISSCIENCE
calcolo delle abbreviazioni del nome e cognome del codice fiscale
'''

import string

VOCALI = [ 'A', 'E', 'I', 'O', 'U' ]
CONSONANTI = list(set(list(string.ascii_uppercase)).difference(VOCALI))

def scomposizione(stringa):
    '''
    scomposizione nelle liste di consonanti e vocali che compongono la stringa
in input
    '''
    stringa = stringa.upper().replace(' ', '')

    consonanti = []
    for car in stringa:
        if car in CONSONANTI:
            consonanti.append(car)

    vocali = [ car for car in stringa if car in VOCALI ]
    return consonanti, vocali
```

```
def abbreviazione(stringa):
    scomposizione_in_consonanti, scomposizione_in_vocali  =
scomposizione(stringa)
    sequenza = scomposizione_in_consonanti
    while len(sequenza) < 3: #se la lunghezza è meno di 3 significa che le
consonanti non bastano e servono le vocali
        try: # pop toglie alla lista il primo elemento e lo restituisce
            sequenza.append(scomposizione_in_vocali.pop(0))
        except: # appende X per stringhe brevi
            sequenza.append('X')
    return ''.join(sequenza[:3]) # trasformazione della lista risultante in
stringa


nomi_da_abbreviare = ['Paolo', 'Riccarco', 'Alessia', 'Luisa', 'Ada', 'Liu',
'Bo']

for nome in nomi_da_abbreviare:
    print (nome, abbreviazione(nome))
```

esercitazione1.py

Esercitazione 2 e 3

scrittura e lettura di un file di testo

--

Scrittura

```
'''
CORSO DI GEOPROCESSIGN - MASTER IN GISSCIENCE
salvataggio di una stringa ad un file di testo
'''

import os

testo = '''
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Ut laoreet sem pellentesque ipsum rutrum consequat. Nunc iaculis tempor aliquet.
Fusce imperdiet pharetra tellus, ut commodo lacus gravida et.
'''

destinazione = r"C:\Users\paolo\Documenti"

text_file = open(os.path.join(destinazione, 'mio_file.txt'), 'w')
text_file.write(testo)
text_file.close()
```

esercitazione2.py

```

```

```
--

## Lettura

```python
'''
CORSO DI GEOPROCESSING - MASTER IN GISSCIENCE
lettura di una stringa da un file di testo
'''

import os
#from esercitazione2 import destinazione
destinazione = r"C:\temp\"


text_file = open(os.path.join(destinazione, 'mio_file.txt'), 'r')
testo = text_file.read()
text_file.close()

print (testo)
```

esercitazione3.py

Esercitazione 4

stampare un'albero di files e directory. uso delle funzioni os:

- os.path()
- os.walk()
- os.listdir()

--

```
'''
CORSO DI GEOPROCESSING - MASTER IN GISSCIENCE
procedura per stampare l'albero di files e directory
'''

import os

dir_sorgente = r"inserire un path"

print ("\nmetodo1")
for root, dirs, files in os.walk(dir_sorgente):
    for file in files:
        print(os.path.join(root,file))

print ("\nmetodo2")
def attraversa_dir(path, livello=0):
    file_e_dir = os.listdir(path)
    for elem in file_e_dir:
        spaziatura =' '*livello*4
        if os.path.isdir(os.path.join(path,elem)):
            print (spaziatura + '['+elem+']')
```
```

```
            attraversa_dir(os.path.join(path,elem), livello=livello + 1)
            print ()
        else:
            print (spaziatura+elem)

attraversa_dir(dir_sorgente)
```

esercitazione4.py

Esercitazione 5

calcolo della lunghezza di un segmento espresso come lista di punti con metodi diversi:

--

```
'''
CORSO DI GEOPROCESSING - MASTER IN GISSCIENCE
calcolo della lunghezza di un segmento espresso come lista di punti
'''
import math

def lunghezza(segmento):
    lunghezza_tot = 0
    for indice in range(0, len(segmento)):
        if indice == 0:
            pass
        else:
            x2 = segmento[indice][0]
            x1 = segmento[indice-1][0]
            y2 = segmento[indice][1]
            y1 = segmento[indice-1][1]
            lunghezza_tot += math.sqrt((x2-x1)**2 + (y2-y1)**2)
    return lunghezza_tot

def lunghezza_elegante(segmento):
    lunghezza_tot = 0
    for indice, p2 in enumerate(segmento):
        if indice == 0:
            pass
        else:
            lunghezza_tot += math.hypot(p2[0]-p1[0], p2[1]-p1[1])
        p1 = p2
    return lunghezza_tot

def lunghezza_pitonica(segmento):
    print (zip(segmento[:-1],segmento[1:])
    diffpt = lambda p: (p[0][0]-p[1][0], p[0][1]-p[1][1])
    lista_diff = map (diffpt, zip(segmento[:-1],segmento[1:]))
    lunghezza_tot = sum(math.hypot(*d) for d in lista_diff)
    return lunghezza_tot

polilinea = [[1.3,3.6],[4.5,6.7],[5.7,6.1],[2.9,0.6],[3.4,2.1],[9.5,2.7]]

print ("lunghezza",lunghezza(polilinea))
print ("lunghezza_elegante",lunghezza_elegante(polilinea))
```

```
print ("lunghezza_pitonica",lunghezza_pitonica(polilinea))

for nome_funzione in ['lunghezza', "lunghezza_elegante", "lunghezza_pitonica"]:
    print (nome_funzione, globals()[nome_funzione](polilinea))
```

esercitazione5.py

ESERCITAZIONE PER CASA

Scrivere una procedura per scansionare il contenuto di una cartella e caricare tutti gli shapefiles in essa contenuti in un unico archivio geopackage. Spunti per lo svolgimento

- scansionare il contenuto di una directory con os.listdir
- estrarre l'estensione da un nome del file con os.path.splitext
- controllare se l'estensione è .shp con if/then e nele caso
    - utilizzare la funzione os.system o subprocess.call per eseguire il comando *ogr2ogr* per la conversione degli shapefiles:
        - `ogr2ogr -append -f GPKG archivio.gpkg file.shp`

Per difficoltà o chiarimenti postare nella sezione ISSUES del repository (https://github.com/enricofer/geoprocessing_giscience_2020)

--

--

MODULI e librerie

Un fattore determinante nel successo di Python è la flessibilità, la modularità e la facile estendibilità permessa dall'organizzazione del codice eseguibile in moduli.

I moduli possono essere:

- predefiniti, già compresi nella dotazione di base del linguaggio
- esterni, contenuti nei path di sistema di Python (PATH, PYTHONPATH). Possono essere preimportati o importati da internet tramite pip/setuptools
- definiti localmente dall'utente in altri files python

Importazione dei moduli

I moduli sono collezioni strutturate ed organizzate di codice python le cui definizioni possono essere importate all'interno di un programma

```
# Importa un modulo con chiamata ad una funzione contenuta
import math
math.floor(10.6)

# Importa singoli elementi da un modulo
from os import path
path.join('C:', 'Utenti', 'paolo', 'documenti')
'C:/Utenti/paolo/documenti'
```

--

Organizzazione dei moduli

Quando importiamo un modulo, Python deve trovare il file corrispondente, e per farlo controlla in ordine le directory elencate nella lista `sys.path`. Una volta trovato il file, Python lo importa, crea un *module object* (oggetto modulo), e lo salva nel dizionario `sys.modules`. Se il modulo viene importato nuovamente in un altro punto del programma, Python è in grado di recuperare il modulo da `sys.modules` senza doverlo importare nuovamente. Inoltre, i moduli `.py` vengono *compilati in bytecode*: un formato più efficiente che viene salvato in file con estensione `.pyc` che a loro volta vengono salvati in una cartella chiamata `__pycache__`. Quando viene eseguito un programma che ha bisogno di importare un modulo, se `modulo.pyc` esiste già ed è aggiornato, allora Python importerà quello invece di ricompilare il modulo in bytecode ogni volta.

--

La libreria standard di Python

Python 3 nella sua versione base già mette a disposizione un set molto ampio di moduli standard sviluppati sotto l'ombrello della comunità di Python: https://docs.python.org/3/library/

Alcuni dei moduli comunemente usati sono i seguenti:

operazioni matematiche: `math`

servizi generici del sistema operativo: `os` `shutil`

interfaccia runtime con il sistema: `sys` `io`

tipi estesi di dato: `time` `datetime` `collections`

protocolli internet e serializzazione: `urllib` `httplib` `json`

archiviazione di dati: `csv` `zipfile`

interfaccia utente: `argparse` `tkinter`

--

`math`

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

```
import math

math.floor(10.6) # troncamento al numero intero inferiore
math.ceil(10.6) # troncamento al numero intero superiore

math.pi # PI greco
math.cos(2*math.pi)
math.degrees(math.pi)

math.isclose(math.sin(0),0) #confronta numeri decimali con approssimazione
```

--

`os` e `os.path`

https://docs.python.org/3/library/os.path.html#os.path.split

```
import os

path = os.path.join("c:/","OSGeo4W64","OSGeo4W.bat")
path = os.path.dirname(__file__)
dirpath = os.path.dirname(path)
os.path.split(path) separa il nome della directory dal file
os.path.splitext(path) separa il file (path completo) dall'estensione

os.remove(path) #cancella il file puntato da path
os.listdir(dirpath) #lista il contenuto di una directory
os.makedirs(path) #crea il path completo
```

--

Uso dei moduli esterni

E' possibile usare moduli esterni configurando opportunamente la variabile d'ambiente PYTHONPATH in modo che essa punti ad una cartella contente il codice del modulo

Oppure si può depositare manualmente la cartella contente il modulo python ed i moduli da esso dipendenti dentro la cartella "site-packages".

La comunità Python ha organizzato un repository dei moduli correntemente utilizzabili da python: https://pypi.python.org/pypi Il repository è utilizzabile tramite il comando pip installabile scaricando ed eseguendo il file get-pip.py

```
python get-pip.py #se non già installato con python
pip install [nome pacchetto]
```

Il comando provvede ad installare il pacchetto desiderato insieme alle sue dipendenze nella versione stabilita dai *"requirements"* del modulo, aiutando il programmatore a superare i conflitti tra versioni diverse dei pacchetti installati.

--

pip install

```
C:\Users\enrico>pip install requests
Collecting requests
...
Collecting idna<3,>=2.5 (from requests)
...
Collecting certifi>=2017.4.17 (from requests)
...
Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 (from requests)
...
Collecting chardet<4,>=3.0.2 (from requests)
...
Installing collected packages: idna, certifi, urllib3, chardet, requests
Successfully installed certifi-2019.11.28 chardet-3.0.4 idna-2.9 requests-2.23.0
urllib3-1.25.8
```

--

altri comandi di pip

```
pip uninstall requests  #disinstallazione pacchetto

pip search requests     #ricerca pacchetto per nome

pip show requests       #mostra dettaglio di un pacchetto

pip list                #lista dei pacchetti installati

pip freeze              #genera in file requirements
```

--

Virtualenv

I Virtualenv (ambienti virtuali) sono un modo semplice per creare progetti Python isolati, in cui installare diverse librerie che non andranno in conflitto con altre librerie in altri ambienti.

Per installare virtualenv sulla proprima macchina, basta digitare su un terminale il comando

```
pip install virtualenv
```

A questo punto, si può creare l'ambiente virtuale, utilizzando il comando

```
cd <directory di progetto>
virtualenv <nome ambiente virtuale> #l'ambiente verrà creato nella directory
corrente
```

--

Attivazione e deattivazione di un virtualenv

```
<path alla directory dell'ambiente virtuale>\scripts\activate.bat #per
l'attivazione
deactivate #per la disattivazione
```

--

Virtualenv wrapper

Uno strumento pratico per gestire gli ambienti virtuali è virtualenvwrapper: https://virtualenvwrapper.readthedocs.io/en/latest/ che permette di individuare un repository comune e permette di gestire la creazione.

vitualenvwrapper funziona nativamente con la shell di Linux e MacOS ma esiste una versione specifica per windows: https://pypi.org/project/virtualenvwrapper-win/

Pipenv

https://pipenv.kennethreitz.org/en/latest/

E' un comando che unifica pip virtualenv e pipfile rendendo possibile la creazione al volo di ambienti di lavoro virtuali e gestendo al contempo le dipendenze ed il loro aggiornamento

--

Anaconda

Anaconda è un sistema di gestione dei pacchetti esterni **alternativo a pip** e si configura come un vero e proprio ambiente operativo che comprende una propria versione di python ed un proprio gestore di ambienti virtuali.

Anaconda è particolarmente adatto per l'installazione di moduli che in pip prevedono la compilazione del codice dal lato utente e quindi per la gestione di moduli esterni in windows dove i requisiti per la compilazione spesso non sono soddisfatti e nei contesti scientifici dove è necessaria la performance e/o la possibilità di utilizzare librerie precompilati, magari in un altro linguaggio.

Anaconda può essere installato da quasta pagina: https://docs.conda.io/projects/conda/en/latest/user-guide/install/windows.html

```
conda install <package>
```

MODULI ESTERNI

requests:

django:

pandas:

jupiter notebooks:

Esercitazione: analizziamo i dati aggiornati del coronavirus:

1) installazione di anaconda: https://docs.conda.io/en/latest/miniconda.html

2) scaricamento del repository dello jupiter notebook di analisi: https://github.com/pdtyreus/coronavirus-ds

3) scaricamento dei dati aggiornati da John Hopkins CSSE: https://github.com/CSSEGISandData/COVID-19

4) installazione delle librerie necessarie:

```
conda install requests
conda install pandas
conda install geopandas
conda install descartes
conda install -c conda-forge jupyterlab
```

5) esecuzione jupiter notebook ed aggiornamento dei dati

COLLABORARE ALLO SVILUPPO DEL SOFTWARE OPEN SOURCE

--

La piattaforma github

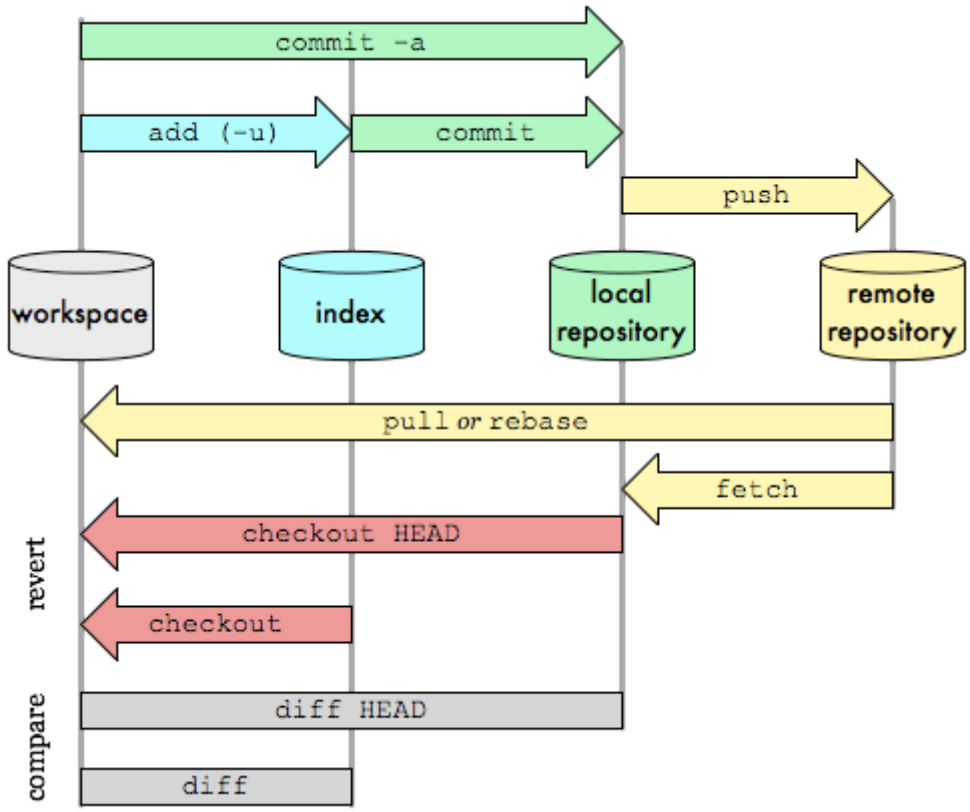
Github (*Ghi-tab* per i non italiani) è una piattaforma web che permette agli sviluppatori di conservare e pubblicare i codici sorgenti dei propri programmi, senza perdere traccia delle versioni del codice stesso. Nel contempo permette a gruppi di sviluppatori di lavorare simultaneamente sullo stesso progetto evitando conflitti.
E' basato su Git, il potente strumento di versionamento creato da Linus Torvarlds, il creatore di Linux, per gestire lo sviluppo collaborativo del sistema operativo open source.

- Git è uno strumento a linea di comando, la cui comprensione esula dagli obiettivi del corso. Per approfondimenti si può fare riferimento ad uno dei tanti tutorial presenti in internet: https://www.slideshare.net/stefanovalle/guida-git
- E' ottimizzato per tenere traccia delle modifiche di file testo a livello di riga (non va bene quindi per file binari o per file di testo con poche righe)

--

Git

per facilitare l'interazione dell'utente, github mette a disposizione Github desktop (disponibile per MacOS e Windows), che permette di gestire con facilità i vari flussi di lavoro senza necessariamente conoscere Git. E' comunque importante comprendere i principi di funzionamento di Git:



--

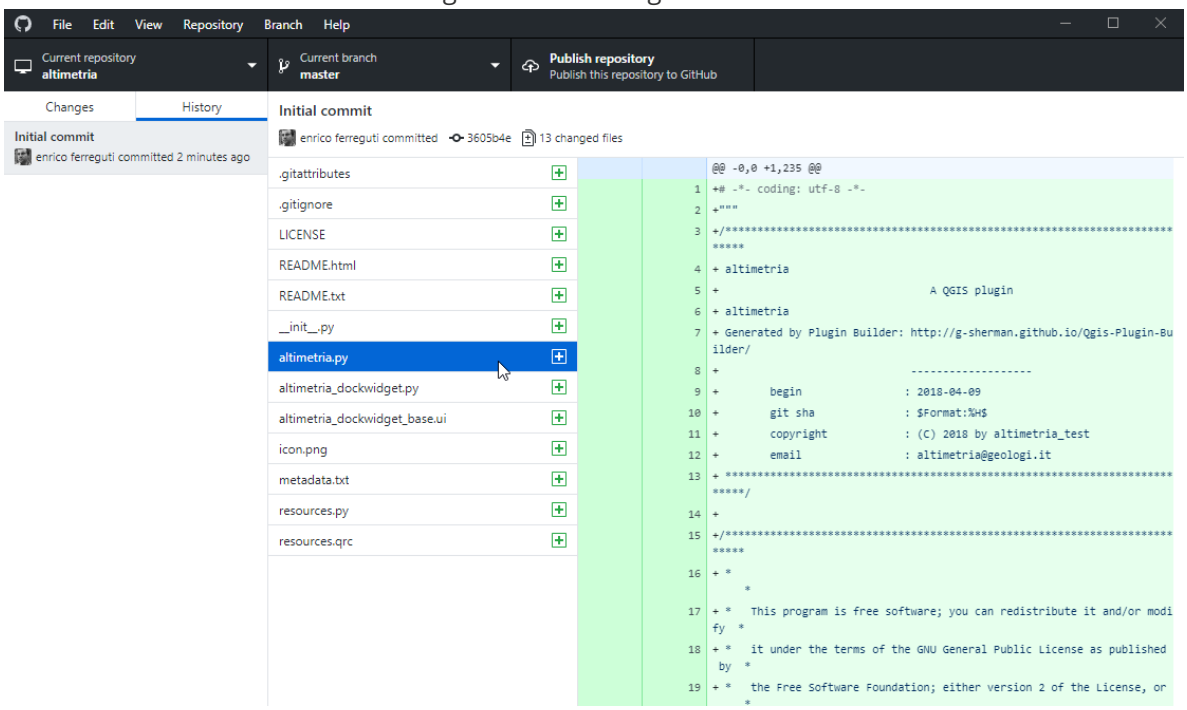
Lessico di git/github

- *repository*: archivio di files e directory gestito da git. può essere locale o remoto
- *commit*: insieme coordinato di modifiche che l'utente registra sul repository
- *branch*: uno stato del repository che viene memorizzato dall'utente separatamente da altri branch. Esistono dei branch di sistema (master, origin etc...) e dei branch utente
- *diff*: operazione che mette in evidenza le modifiche di riga tra branch
- *merge*: operazione che permette di fondere tra loro due branch (per esempio un branch di sviluppo nel branch master) mettendo in evidenza eventuali conflitti
- *clone*: operazioone di clonazione in locale di un repository remoto
- *push*: operazione con la quale si si conferisce (submit) un branch locale ad un repository remoto tenendo condo dei conflitti tra versioni
- *pull*: operazione con la quale si scarica in locale un repository remoto
- *pull request*: operazione con la quale un utente propone la modifica di un repository

--

github desktop

E' fondamentalmente un'interfaccia grafica di Git integrata con il servizio di Github



--

Esercitazione di Github

- accreditarsi su Github, scaricare ed installare Github desktop
- inizializzare un repository
- clonare un repository
- modificare i files / verificare le differenze / realizzare un commit
- creare un branch
- fondere (merge) due branch
- pubblicare un repository
- inviare una pull request (PR)