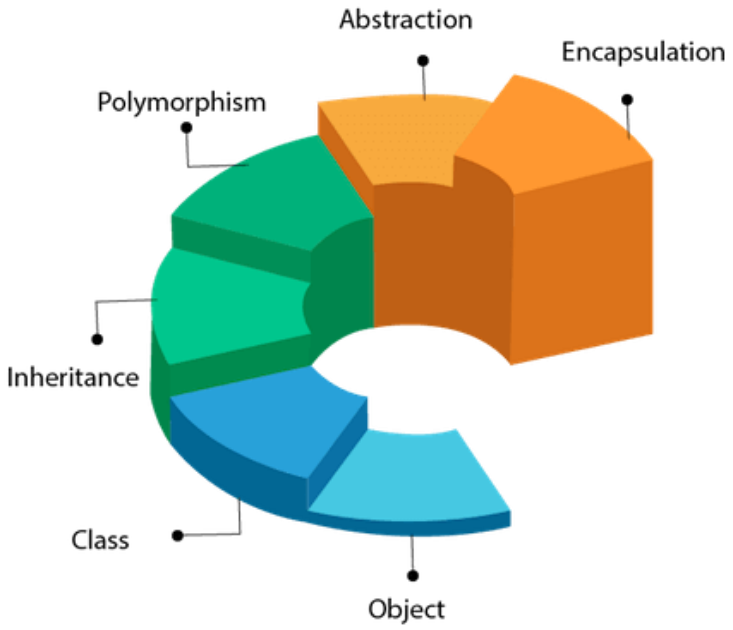
Object Oriented Programming

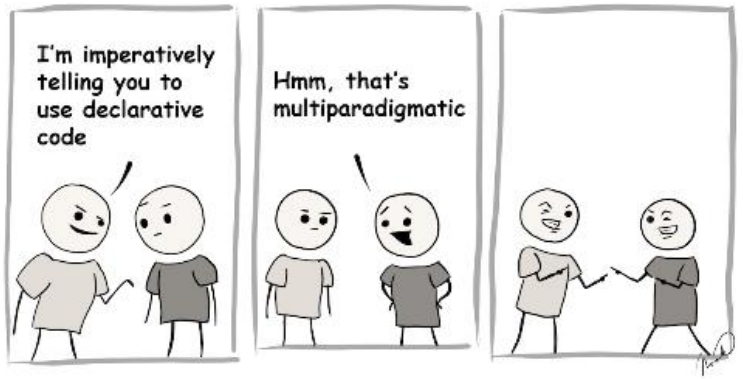


Object Oriented Programming

- paradigmi di programmazione
  - OOP
- oggetto
- classe
  - attributi
  - costruttore
  - metodi
- tuple

paradigma di programmazione

- il ***paradigma di programmazione*** definisce il modo in cui il programmatore concepisce il programma
- i vari paradigmi si ***differenziano***
- per le ***astrazioni*** usate per rappresentare gli elementi di un programma (funzioni, oggetti, variabili …)
- per i ***procedimenti*** usati per l'elaborazione dei dati (assegnamento, iterazione, gestione del flusso dei dati ...)



paradigmi

- paradigma ***imperativo***
  - programmazione procedurale (‘60)
  - programmazione strutturata (‘60-’70)
- programmazione orientata agli ***eventi***
  - *interfacce grafiche*
- programmazione ***logica***
  - *intelligenza artificiale*
- programmazione ***funzionale***
  - *applicazioni matematiche e scientifiche*
- programmazione ***orientata agli oggetti***

paradigma imperativo

- il programma viene inteso come un insieme di istruzioni (direttive o ***comandi***) ogni istruzione è un "***ordine***" che viene impartito al computer
- linguaggi imperativi:
  - *Cobol, C, Basic, Pascal, Fortran…*

```
utente = input("Come ti chiami?")
if utente == "Alberto"
   print("buongiorno prof")
else
   print("ciao", utente)
```

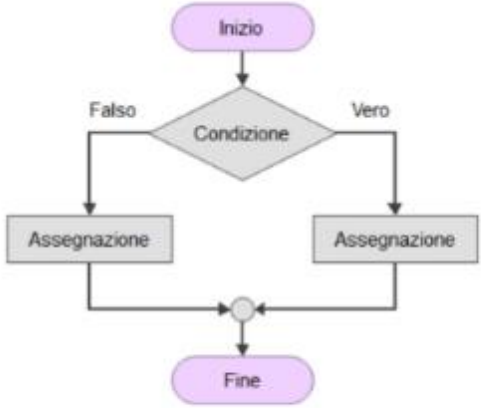
paradigma procedurale

- paradigma basato su ***funzioni*** (*procedure*) che contengono blocchi di programma riutilizzabili
- le funzioni possono avviare altre funzioni o riavviare se stesse
- la ***scomposizione*** in funzioni agevola lo sviluppo, il collaudo e la gestione dei programmi
- ogni funzione risolve un problema specifico ogni volta che si presenta nel programma
  - può essere scritta una volta sola e riutilizzata più volte
- la maggior parte dei linguaggi più diffusi consentono la programmazione procedurale

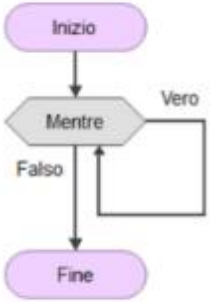
programmazione strutturata

- strutture di controllo:
  - ***sequenza***
  - ***selezione***
  - ***iterazione***

SEQUENZA



SELEZIONE



ITERAZIONE

Qualunque algoritmo può essere implementato utilizzando queste tre sole strutture (Teorema di Böhm-Jacopini, 1966)

programmazione orientata agli eventi

- in un programma ***tradizionale*** l'esecuzione delle istruzioni segue ***percorsi fissi***, che si ramificano soltanto in punti predefiniti dal programmatore
- nella ***programmazione orientata agli eventi*** il flusso del programma è determinato dal verificarsi di ***eventi esterni***
  - il programma attende che accadano determinati eventi e, quando ciò avviene, avvia la sequenza specificata
- gli eventi possono essere azioni dell'utente, input proveniente da un sensore, messaggi di altri sistemi informatici …
- linguaggi orientati agli eventi:
  - JavaScript, Scratch …

```
<input type="button" value="Clicca qui!" onClick="showMessage();">
```

programmazione logica

- la ***programmazione logica*** adotta la logica del primo ordine
  - per rappresentare
  - per elaborare l'informazione
- richiede al programmatore di ***descrivere*** la struttura logica del problema piuttosto che il modo di ***risolverlo***
- linguaggi: ProLog …

esempio prolog

```
padre(pippo,gino).
padre(luca,pippo).
padre(francesco,luca).
padre(pippo,manuela).
padre(luca,genoveffa).
padre(luca,giuseppina).

madre(giulia,gino).
madre(lina,pippo).
madre(gelda,luca).
madre(giulia,manuela).
madre(lina,genoveffa).
madre(franca,giuseppina).

genitore(X,Y) :- padre(X,Y).
genitore(X,Y) :- madre(X,Y).

antenato(X,Y) :- genitore(X,Y).
antenato(X,Y) :- genitore(X,Z), antenato(Z,Y).

fratello_sorella(X,Y) :-
padre(Z,X), padre(Z,Y), madre(W,X), madre(W,Y), X \= Y.
fratellastro_sorellastra(X,Y) :-
genitore(W,X), genitore(W,Y), X \= Y.
zio_zia(X,Y) :- genitore(Z,Y), fratello_sorella(X,Z).
```

```
?- madre(X,gino).
X = giulia ;
```

```
?- fratello_sorella(X,Y).

X = gino
Y = manuela ;

X = pippo
Y = genoveffa ;

X = manuela
Y = gino ;

X = genoveffa
Y = pippo ;
```

programmazione funzionale

- nella ***programmazione funzionale*** il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche
- principale vantaggio:
  - mancanza di effetti collaterali (side-effect)
  - più facile verifica della correttezza e eliminazione di bug del programma
  - facilita la programmazione parallela
- linguaggi: LISP, Logo, Haskell …

esempio Haskell

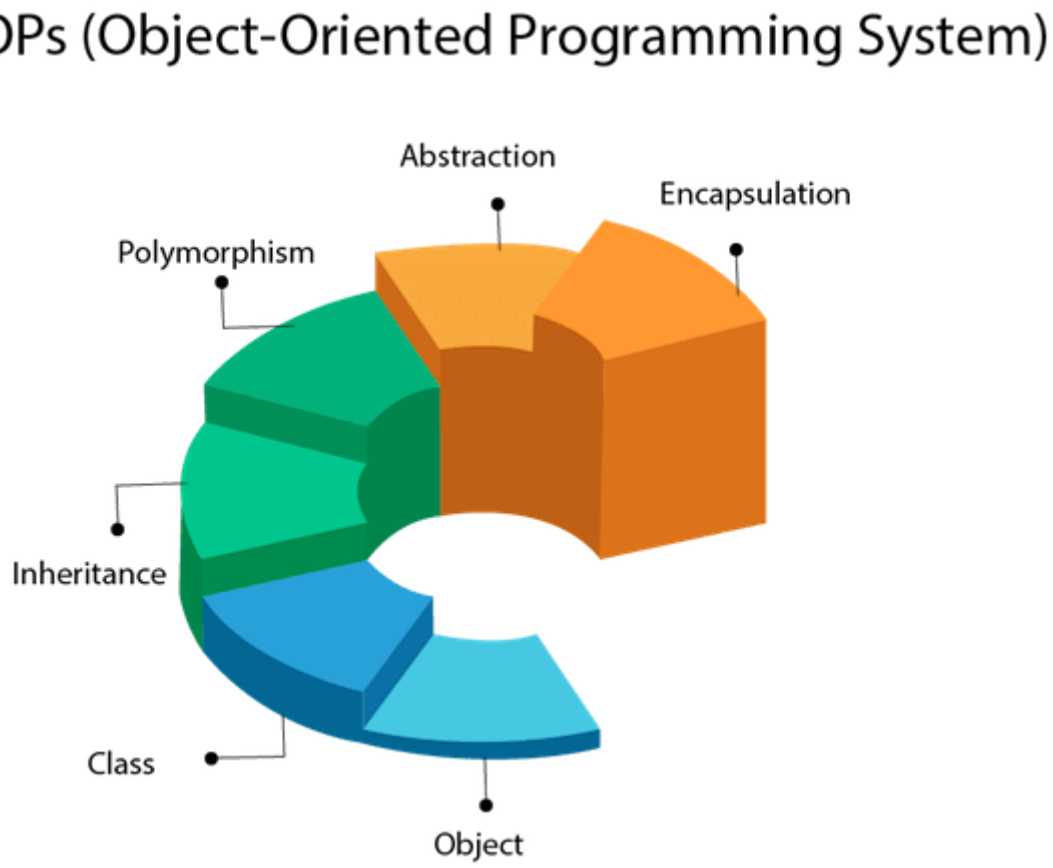
```
def create_pow(exponent: float):
    """Create and return a function, which calculates a power.
    """
    def result(base: float):
        return base ** exponent
    return result

root = create_pow(0.5)
cube = create_pow(3)

print(root(3))
print(root(4))

print(cube(3))
print(cube(4))
```

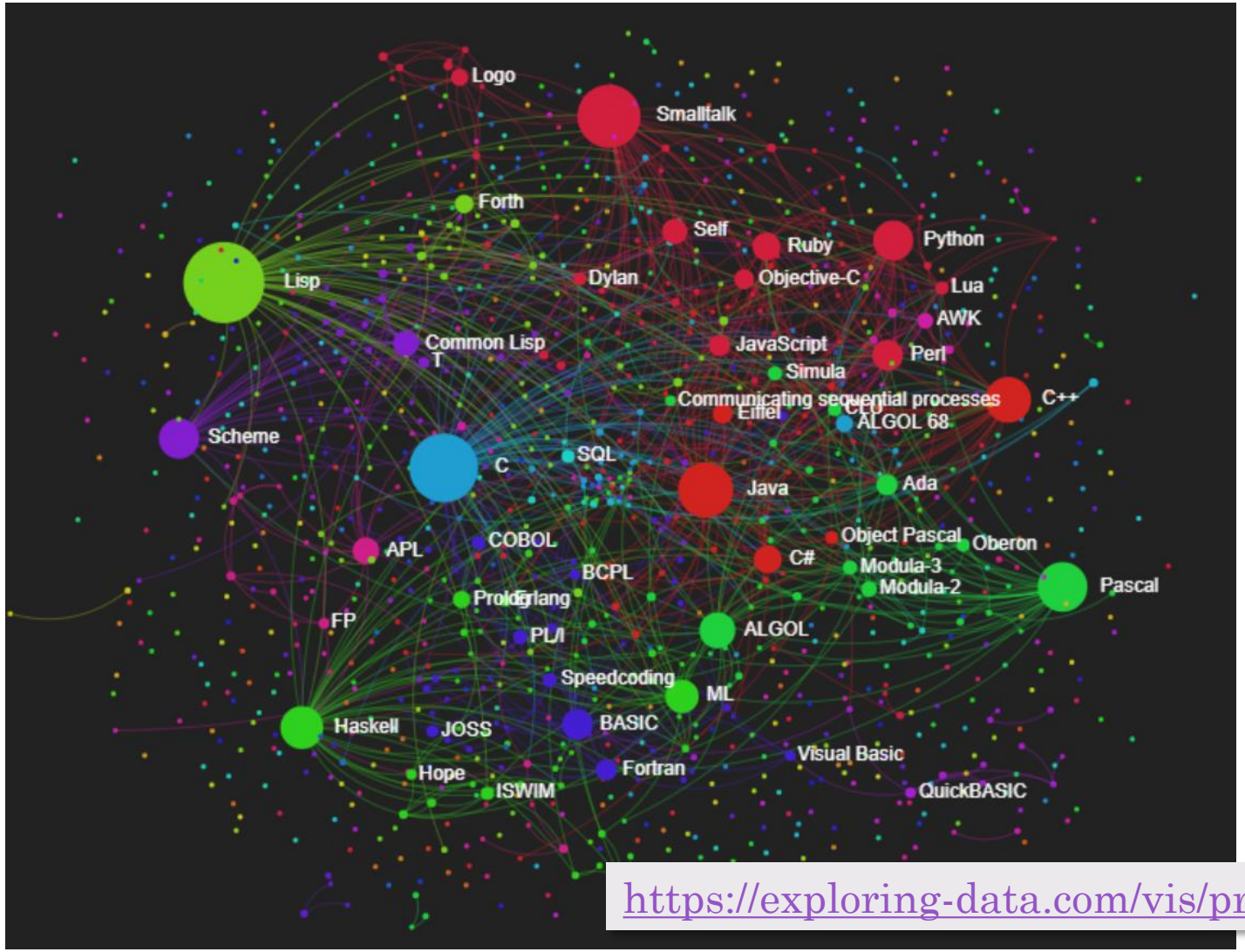
programmazione orientata agli oggetti



Object Oriented Programming

- la ***programmazione orientata agli oggetti*** (***OOP***, ***O***bject ***O***riented ***P***rogramming) permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi

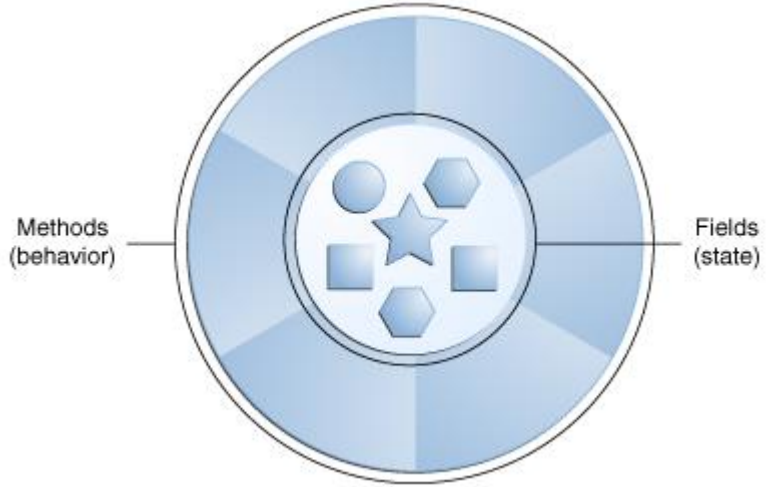
influenze fra linguaggi



https://exploring-data.com/vis/programming-languages-influence-network/

oggetto

- analisi della realtà e definizione del ***dominio applicativo***
  - evidenziare informazioni essenziali eliminando quelle non significative per il problema
- un ***oggetto*** rappresenta un oggetto fisico o un concetto del dominio
  - memorizza il suo ***stato*** interno in campi privati (attributi dell'oggetto)
    - concetto di ***incapsulamento*** (black box)
  - offre un insieme di ***servizi***, come ***metodi*** pubblici (comportamenti dell'oggetto)
- realizza un ***tipo di dato astratto***
  - (ADT - *Abstract Data Type*)



esempio: automobile di Mario

- nome dell'oggetto: «AutoDiMario»
- classe: ***Automobile***
- ***attributi***
  - velocità_massima → 220
  - velocità_attuale → 120
  - tipo_freni → tamburo
- ***metodi***
  - accelera(n) → aumenta velocità nKm/h
  - frena()

classi e oggetti

- ogni oggetto ha una ***classe*** di origine (*è istanziato da una classe*)
- la classe definisce la stessa ***forma iniziale*** (campi e metodi) a tutti i suoi oggetti
- ma ogni oggetto
  - ha la sua ***identità***
  - ha uno stato e una locazione in memoria distinti da quelli di altri oggetti
    - sia istanze di classi diverse che della stessa classe

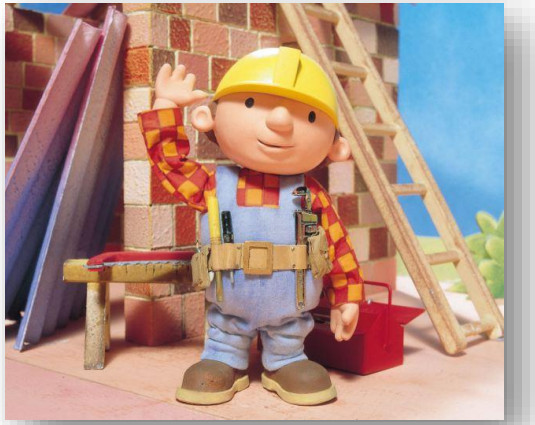
classe Automobile

oggetti e classi

- da una ***classe*** possono essere istanziati (creati) ***più oggetti***
- tutti con gli ***stessi attributi***
  - ma ognuno ha una sua ***valorizzazione*** degli attributi
- tutti hanno lo stesso ***comportamento***
  - rispondono ai messaggi attivando i ***metodi*** della classe

costruzione di oggetti

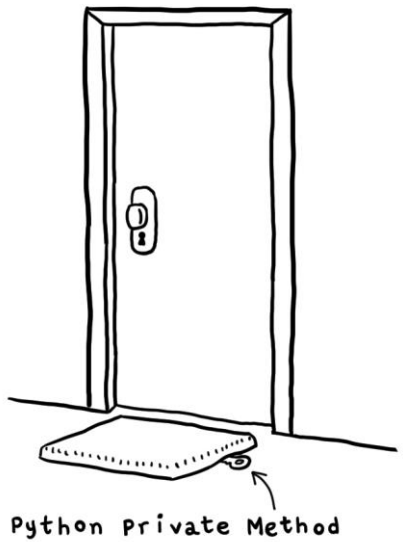
- costruzione di oggetti (***istanziazione***)
- `__init__`: metodo ***costruttore***
- eseguito ***automaticamente*** alla creazione di un oggetto
  - *instantiation is initialization*
- `self`: primo parametro di tutti i metodi
  - non bisogna passare un valore esplicito
  - rappresenta l'oggetto di cui si chiama il metodo
  - permette ai metodi di accedere ai campi



definizione della classe

```
class Automobile:

    def __init__(self, vm: int, va: int, freno: str):
        '''
        vm velocità massima
        va velocità attuale
        freno tipo freno
        '''
        self._vm = max(vm,0)
        self._va = max(va,0)
        if freno == 'tamburo':
            self._freno = 'tamburo'
        else:
            self._freno = 'disco'
```



i metodi

```
    def accelera(self,km: int):
        ''' aumenta la velocià di km chilometri orari'''
        if km <= 0:
            return
        self._va += km
        if self._va > self._vm:
            self._va = self._vm

    def frena(self):
        ''' dipende dal tipo dei freni '''
        if self._freno == 'disco':
            self._va -= 20
        else:
            self._va -= 10
        self._va = max(self._va,0)
```

```
    def to_tuple(self) -> tuple:
        return self._vm, self._va, self._freno

    def to_string(self) -> str:
        s = 'velocità massima: ' + str(self._vm)
        s += ' velocità attuale: ' + str(self._va)
        s += ' freni a ' + self._freno
        return s
```

definizioni

- ***campi***: memorizzano i ***dati caratteristici*** di una istanza
  - ogni pallina ha la sua posizione (x, y) e la sua direzione (dx, dy)
- ***parametri***: ***passano*** altri ***valori*** ad un metodo
  - se alcuni dati necessari non sono nei campi
- ***variabili locali***: memorizzano ***risultati parziali***
  - generati durante l'elaborazione del metodo
  - nomi ***cancellati*** dopo l'uscita dal metodo
- ***variabili globali***: definite ***fuori*** da tutte le funzioni
  - usare sono se strettamente necessario
  - meglio avere qualche parametro in più, per le funzioni

in Python tutto è un oggetto

```
n = 15                              # viene istanziato l'oggetto n della classe int
print('tipo di n ',type(n))                             # type funzione
print('rappresentazione binaria di n ',bin(n))  # bin funzione
print('cifre binarie di n ',n.bit_length())     # bit_length metodo
```

```
tipo di n <class 'int'>
rappresentazione binaria di n 0b1111
cifre binarie di n 4
```

```
f = 3.25                                                # f oggetto della classe float
print('tipo di f ',type(f))                             # type funzione
print(f.as_integer_ratio(),'frazione generatrice di',f)  # as_integer_ratio metodo
```

```
tipo di f <class 'float'>
(13, 4) frazione generatrice di 3.25
```

esercizi

esercizi (1)

classe ellisse

- definire una ***classe*** che modella un'ellisse
- ***campi privati*** (parametri del costruttore)
  - semiassi: a, b
- ***metodi pubblici*** per ottenere:
  - area: $\pi \cdot a \cdot b$
  - distanza focale: $2 \cdot \sqrt{|a^2 - b^2|}$
- nel corpo principale del ***programma***
  - creare un oggetto con dati forniti dall'utente
  - visualizzare area e distanza focale dell'ellisse