INDICE

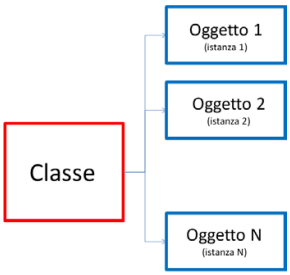
Definire una classe

In generale:

- Definire una `classe` vuol dire creare un nuovo tipo di dati.
- Quando si parla di `oggetti` si intendono `istanze` di una classe (1,2 3 .. sono oggetti di tipo int)

Sulla definizione di una classe:

- Il nuovo tipo di dati potra' essere caratterizzato sia da attributi (dati propriamente detti) sia da metodi (funzioni)
- Definire una classe vuol dire quindi definire sia la struttura dei dati sia la logica dei metodi
- Ciascun oggetto puo' essere pensato come ad una macchina a stati in cui i dati definiscono lo stato ed i metodi le dinamiche di evoluzione.



Progettiamo una classe v1

Progettiamo una semplice classe:

- che rappresenti un libro
- che abbia i seguenti attributi:
  - Titolo (stringa)
  - Autore (stringa)
  - Numero di pagine
- che abbia i seguenti metodi

- Info() : restituisce una stringa con le informazioni relative al libro

in modo che si possa usare nel seguente modo:

In [ ]:
```
#creo un istanza della classe (un oggetto)
l1 = Libro(titolo="Promessi sposi",autore="Manzoni",pagine=500)
#uso un metodo dell'oggetto
l1.info()

#creo un'altra istanza della classe (un oggetto)
l2 = Libro()
#uso un metodo dell'oggetto l2
l2.info()
```

Vediamo l'implementazione:

In [3]:
```
class Libro:
    ##costruttore della classe
    def __init__(self,titolo="non specificato",autore="non specificato",pagine=0):
        #gli attributi sono definiti nel costruttore
        self.titolo = titolo
        self.autore = autore
        self.pagine = pagine
    def info(self):
        print('Autore: {0}, Titolo : {1}, Pagine:
{2}'.format(self.titolo,self.autore,self.pagine))

#creo un istanza della classe (un oggetto)
l1 = Libro(titolo="Promessi sposi",autore="Manzoni",pagine=500)
#uso un metodo dell'oggetto
l1.info()

#creo un'altra istanza della classe (un oggetto)
l2 = Libro()
#uso un metodo dell'oggetto l2
l2.info()
```

```
Autore: Promessi sposi, Titolo : Manzoni, Pagine: 500
Autore: non specificato, Titolo : non specificato, Pagine: 0
```

Significato di self

Quando creo il nuovo oggetto con

In [4]:
```
l1 = Libro(titolo="Promessi sposi",autore="Manzoni",pagine=500)
```

L'interprete lo traduce in

In [5]:
```
Libro.__init__(self = l1 ,titolo="Promessi sposi",autore="Manzoni",pagine=500)
```

Analogamente, quando uso la sintassi:

In [6]:
```
l1.info
```

Out[6]:
```
<bound method Libro.info of <__main__.Libro object at 0x02D9EB70>>
```

L'interprete la traduce in

In [7]:
```
Libro.info(self = l1)
```

```
Autore: Promessi sposi, Titolo : Manzoni, Pagine: 500
```

print di un libro

La stringa stampata su stdout dal metodo info e' una buon candidata anche per una applicazione della print a un oggetto di tipo Libro:

In [8]:
```
class Libro:
    ##costruttore della classe
    def __init__(self,titolo="non specificato",autore="non specificato",pagine=0):
        #gli attributi sono definiti nel costruttore
        self.titolo = titolo
        self.autore = autore
        self.pagine = pagine

    def info(self):
        print('Autore: {0}, Titolo : {1}, Pagine:
{2}'.format(self.titolo,self.autore,self.pagine))

    def __str__(self):
        return 'Autore: {0}, Titolo : {1}, Pagine:
{2}'.format(self.titolo,self.autore,self.pagine)
```

In [9]:
```
l2 = Libro()
print(l2)
```

```
Autore: non specificato, Titolo : non specificato, Pagine: 0
```

Progettiamo una classe v2

Passiamo adesso ad una classe con una logica un po' piu' complessa

Definiamo le caratteristiche desiderate della nostra classe:

- si chiamera' IntPair
- i principali attributi saranno due interi a e b
- vogliamo costruire un IntPair con la seguente sintassi: IntPair(a,b) (con a e b non specificati, a e b entrambi 1)
- vogliamo che ciascun oggetto IntPair sia rappresentato dalla stringa ip(a,b)
- vogliamo definire una funzione mul che restituisce il prodotto tra a e b

Inoltre

- vogliamo definire i seguenti operatori:
  - ip(a1,b1) + ip(a2,b2) = ip(a1+a2 , b1+b2)
  - ip(a1,b1) - ip(a2,b2) = ip(a1-a2 , b1-b2)
  - ip(a1,b1) * ip(a2,b2) = ip(a1*a2 , b1*b2)
- vogliamo definire le seguenti funzioni che eseguono in place le operazioni corrispondenti agli operatori precedenti
  - ip(a1,b1).sum(a2,b2) -in-place-> ip(a1+a2 , b1+b2)
  - ip(a1,b1).sub(a2,b2) -in-place-> ip(a1-a2 , b1-b2)
  - ip(a1,b1).mul(a2,b2) -in-place-> ip(a1*a2 , b1*b2)

Definiamo la nostra classe

una classe vuota

In [10]:
```
#primissima versione: classe vuota
class IntPair:
    pass
```

Adesso uso la classe:

In [11]:
```
#uso la classe
a = IntPair()#creo un oggetto di tipo IntPair e gli associo un nome

print(IntPair)
print(type(a))
print(a)
```

```
print('-'*10)

print(int)
print(type(1))
```

```
<class '__main__.IntPair'>
<class '__main__.IntPair'>
<__main__.IntPair object at 0x02DA3950>
----------
<class 'int'>
<class 'int'>
```

- è possibile usare una classe vuota
- NB distinzione tra oggetto e classe

aggiungiamo una docstring, un costruttore, una funzione di stampa ed il metodo 'mul'

In [12]:
```
#definisco la classe
class IntPair:
    """classe intpair"""
    def __init__(self , a=1 , b=1):
        """costruttore classe Intpair"""
        self.a = a
        self.b = b

    def __str__(self):
        return 'ip({0},{1})'.format(self.a,self.b)

    def mul(self):
        return self.a*self.b
```

uso la classe appena creata

In [13]:
```
#invoco il costruttore __init__
i=IntPair()      # IntPair.__init__(self=i , a=1  , b=1 )
j=IntPair(10,10) # IntPair.__init__(self=i , a=10 , b=10)
#
print('i = ' , i)
print('j = ' , j)
```

```
i =  ip(1,1)
j =  ip(10,10)
```

In [14]:
```
#uso mul
print('j.mul() =',j.mul())
```

```
j.mul() = 100
```

- init e str sono cosiddetti metodi 'dunder' (double underscore) che implementano funzionalita' standard della classe:
  - init viene usata quando si usa il nome della classe per creare un nuovo oggetto
  - str si usa quando c'e' necessita' di convertire in stringa un oggetto
- il primo parametro (che per convenzione si chiama self) dei metodi si riferisce all'istanza dell'oggetto stesso
- nel costruttore a e self.a sono due 'cose' diverse

Accedo direttamente ad a e b

In [15]:
```
j.a,j.b
```

Out[15]: (10, 10)

se vogliamo rendere a e b non direttamente accessibili....

```
```

In [16]:
```
class IntPair:
    """classe intpair"""
    def __init__(self , a=1 , b=1):
        """costruttore classe Intpair"""
        self.__a = a
        self.__b = b

    def __str__(self):
        return 'ip({0},{1})'.format(self.__a,self.__b)

    def mul(self):
        return self.__a*self.__b
```

Gli attributi sono stati mascherati ma sono comunque raggiungibili

In [17]:
```
j=IntPair(10,20)
#j.__a
print(dir(j))
```

```
['_IntPair__a', '_IntPair__b', '__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'mul']
```

aggiungo degli operatori

In [18]:
```
class IntPair:
    """classe intpair"""
    def __init__(self , a=1 , b=1):
        """costruttore classe IntPair"""
        self.a = a
        self.b = b

    def __str__(self):
        return 'ip({0},{1})'.format(self.a,self.b)

    #operatori binari standard
    def __add__(self,other):
        return IntPair(self.a+other.a,self.b+other.b)

    def __sub__(self,other):
        return IntPair(self.a-other.a,self.b-other.b)

    def __mul__(self,other):
        return IntPair(self.a*other.a,self.b*other.b)


    #operatori in place
    def add(self,a,b):
        self.a = self.a + a
        self.b = self.b + b

    def sub(self,a,b):
        self.a = self.a - a
        self.b = self.b - b

    def mul(self,a,b):
        self.a = self.a * a
        self.b = self.b * b


i20 = IntPair(20,10)
i2 = IntPair(2,1)
print('{0} + {1} = {2}'.format(i20,i2,i20+i2))
print('{0} - {1} = {2}'.format(i20,i2,i20-i2))
print('{0} * {1} = {2}'.format(i20,i2,i20*i2))
```

```
i20.add(1,1)
print(i20)
```

```
ip(20,10) + ip(2,1) = ip(22,11)
ip(20,10) - ip(2,1) = ip(18,9)
ip(20,10) * ip(2,1) = ip(40,10)
ip(21,11)
```

In teoria a e b dovevano essere interi ma ....

funziona anche coi float

In [19]:
```
a=IntPair(3.0,2.456)
b=IntPair(5,6)
#NB: operazioni miste
print('{0} + {1} = {2}'.format(a,b,a+b))
print('{0} - {1} = {2}'.format(a,b,a-b))
print('{0} * {1} = {2}'.format(a,b,a*b))
```

```
ip(3.0,2.456) + ip(5,6) = ip(8.0,8.456)
ip(3.0,2.456) - ip(5,6) = ip(-2.0,-3.544)
ip(3.0,2.456) * ip(5,6) = ip(15.0,14.736)
```

e coi complessi

In [20]:
```
a=IntPair(3.0,2.456)
b=IntPair(5+2j,6-8j)
#NB: operazioni miste
print('{0} + {1} = {2}'.format(a,b,a+b))
print('{0} - {1} = {2}'.format(a,b,a-b))
print('{0} * {1} = {2}'.format(a,b,a*b))
```

```
ip(3.0,2.456) + ip((5+2j),(6-8j)) = ip((8+2j),(8.456-8j))
ip(3.0,2.456) - ip((5+2j),(6-8j)) = ip((-2-2j),(-3.544+8j))
ip(3.0,2.456) * ip((5+2j),(6-8j)) = ip((15+6j),(14.736-19.648j))
```

- Ci sono molti altri operatori da potere ridefinire per una classe, per un elenco vedi qui
- che io sappia non è possibile definire ex-novo (simbolo + logica) nuovi operatori

Esercizio

v2d_0.py

Costruire una classe V2D che gestisce vettori in due dimensioni in coordinate cartesiane applicati nell'origine:

- le coordinate cartesiane della fine del vettore sono x e y (dati)
- implementa la somma (+, __sum) e la differenza vettoriale (-, __sub)
- implementa il prodotto scalare (*, __mul)
- implementa il metodo per la determinazione del modulo del vettore (__abs)
- supporta la conversione a stringa con formato '(x,y)' __str
- implementa il modulo per il test di uguaglianza tra vettori (__eq) (attenzione ai float, gestire in modo semplice la tolleranza ad esempio ponendola pari a 1e-6 nel codice)
- implementa il modulo per il test di disuguaglianza tra vettori (__eq__)

una volta implementata la classe potra' essere usata nel seguente modo:

In [21]:
```
#la classe V2D è implementata in v2d_0.py
from v2d_0 import V2D

#il costruttore accetta due parametri
k=V2D(20,30)
v=V2D(2,3)
```

```
#si definisce come ciascun vettore è stampato
print('k = ',k)
print('v = ',v)
#si definisce l'operatore di somma
print('{0} + {1} = {2}'.format(k,v,k+v))
#sottrazione
print('{0} - {1} = {2}'.format(k,v,k-v))
#moltiplicazione
print('{0} * {1} = {2}'.format(k,v,k*v))
#valore assoluto
print('abs(v)  ',abs(v))
#uguaglianza
print('v==k  ',v==k)
print('v==v  ',v==v)

#implementare anche un semplice sistema per gestire la tolleranza
#nei test di uguaglianza
print('-'*40)
v2=V2D(2.0001,3.0001)
v3=V2D(2.00000001,3.00000001)
print('v = ',v)
print('v2 = ',v2)
print('v3 = ',v3)
print('v==v2  ',v==v2)
print('v==v3  ',v==v3)
```

```
k =  (20,30)
v =  (2,3)
(20,30) + (2,3) = (22,33)
(20,30) - (2,3) = (18,27)
(20,30) * (2,3) = 130
abs(v)   3.60555127546
v==k   False
v==v   True
----------------------------------------
v =  (2,3)
v2 =  (2.0001,3.0001)
v3 =  (2.00000001,3.00000001)
v==v2   False
v==v3   True
```

v2d_1.py

una versione con una gestione della tolleranza piu'avanzata:

In [22]:
```
from v2d_1 import V2D
k=V2D(20,30)
v=V2D(2,3)
print('{0} + {1} = {2}'.format(k,v,k+v))
print('{0} - {1} = {2}'.format(k,v,k-v))
print('{0} * {1} = {2}'.format(k,v,k*v))
print('abs(v)  ',abs(v))
print('v==k  ',v==k)
print('v==v  ',v==v)

print('-'*40)
v2=V2D(2.0001,3.0001)
v3=V2D(2.00000001,3.00000001)
print('v = ',v)
print('v2 = ',v2)
print('v3 = ',v3)

print('-'*20)
V2D.set_tol(1e-2)
print('tolleranza = {0}'.format(V2D.get_tol()))
print('v==v2  ',v==v2)
print('v==v3  ',v==v3)
```

```
print('-'*20)
V2D.set_tol(1e-5)
print('tolleranza = {0}'.format(V2D.get_tol()))
print('v==v2  ',v==v2)
print('v==v3  ',v==v3)

print('-'*20)
V2D.set_tol(1e-10)
print('tolleranza = {0}'.format(V2D.get_tol()))
print('v==v2  ',v==v2)
print('v==v3  ',v==v3)
```

```
(20,30) + (2,3) = (22,33)
(20,30) - (2,3) = (18,27)
(20,30) * (2,3) = 130
abs(v)   3.60555127546
v==k   False
v==v   True
----------------------------------------
v =  (2,3)
v2 =  (2.0001,3.0001)
v3 =  (2.00000001,3.00000001)
--------------------
tolleranza = 0.01
v==v2   True
v==v3   True
--------------------
tolleranza = 1e-05
v==v2   False
v==v3   True
--------------------
tolleranza = 1e-10
v==v2   False
v==v3   False
```

Costruiamo una classe derivata

Vogliamo costruire una classe che abbia le stesse caratteristiche dell'IntPair alle quali si ne aggiungono alcune.

- un fattore k da moltiplicare a e b
- una funzione di stampa che permetta la distinzione dagli IntPair

Si dice che IntPairK `eredita` da IntPair.

In [23]:
```
class IntPairK(IntPair):
    def __init__(self,a=1,b=1,k=1):
        IntPair.__init__( self , k*a , k*b )
        self.k=k

    def __str__(self):
        return 'ipk<{0},{1}>'.format(self.a,self.b)
```

usiamo la classe

In [24]:
```
a=IntPairK(1,2,3)
b=IntPair(4,5)
print('{0} + {1} = {2}'.format(a,b,a+b))
```

```
ipk<3,6> + ip(4,5) = ip(7,11)
```

si nota che la somma tra IntPair e IntPairK restituisce comunque un IntPair (perché usiamo comunque l'operatore `add` di IntPair)

- vogliamo che la somma mista ci restituisca un IntPairK con il k dell'unico IntPairK
- vogliamo che la tra due IntPairK restituisca un IntPairK con k pari al prodotto dei due k

In [25]:
```
class IntPairK(IntPair):
```

```
    def __init__(self,a=1,b=1,k=1):
        IntPair.__init__(self,k*a,k*b)
        self.k=k

    def __str__(self):
        return 'ipk<{0},{1}>'.format(self.a,self.b)


    def __add__(self,other):
        try:
            getattr(other,'k')
        except AttributeError:
            return IntPairK(self.a+other.a,self.b+other.b,self.k) ## ipk + ip
        else:
            return IntPairK(self.a+other.a,self.b+other.b,self.k*other.k) ## ipk + ipk

a=IntPairK(1,2,3)
b=IntPair(4,5)
print('{0} + {1} = {2}'.format(b,b,b+b))
print('{0} + {1} = {2}'.format(a,a,a+a))
print('{0} + {1} = {2}'.format(a,b,a+b))
print('{0} + {1} = {2} !!!!!'.format(b,a,b+a))
```

```
ip(4,5) + ip(4,5) = ip(8,10)
ipk<3,6> + ipk<3,6> = ipk<54,108>
ipk<3,6> + ip(4,5) = ipk<21,33>
ip(4,5) + ipk<3,6> = ip(7,11) !!!!!
```

ancora c'e' qualcosa che non va

In [26]:
```
class IntPairK(IntPair):
    def __init__(self,a=1,b=1,k=1):
        IntPair.__init__(self,k*a,k*b)
        self.k=k

    def __str__(self):
        return 'ipk<{0},{1}>'.format(self.a,self.b)


    def __add__(self,other):
        try:
            getattr(other,'k')
        except AttributeError:
            return IntPairK(self.a+other.a,self.b+other.b,self.k) ## ipk + ip
        else:
            return IntPairK(self.a+other.a,self.b+other.b,self.k*other.k) ## ipk + ipk

    #occhio a cosa succede se non metto questo
    def __radd__(self,other):
        return IntPairK(self.a+other.a,self.b+other.b,self.k) ## ip + ipk

a=IntPairK(1,2,3)
b=IntPair(4,5)
print('{0} + {1} = {2}'.format(b,b,b+b))
print('{0} + {1} = {2}'.format(a,a,a+a))
print('{0} + {1} = {2}'.format(a,b,a+b))
print('{0} + {1} = {2}'.format(b,a,b+a))
```

```
ip(4,5) + ip(4,5) = ip(8,10)
ipk<3,6> + ipk<3,6> = ipk<54,108>
ipk<3,6> + ip(4,5) = ipk<21,33>
ip(4,5) + ipk<3,6> = ipk<21,33>
```

Esercizio

data la classe

In [27]:
```
class FiguraGeometrica:
    def area(self):
        raise NotImplementedError('metodo area non implementato')
    def perimetro(self):
        raise NotImplementedError('metodo area non implementato')
```

implementare le classi derivate Cerchio, Rettangolo, Quadrato nel modo che si ritiene piu' opportuno implementando i metodi area, perimetro e `str` in modo che il codice si comporti nel seguente modo:

In [28]:
```
from figure import FiguraGeometrica,Cerchio,Quadrato,Rettangolo

listafigure = [FiguraGeometrica(),
                   Cerchio(1.0),
                   Quadrato(1.0),
                   Rettangolo(1.0,2.0)]
for f in listafigure:
    print('-'*20)
    print(f)
    try:
        f.area()
        f.perimetro()
    except Exception as err:
        print(err)
```

```
--------------------
Figura geometrica generica
metodo area non implementato
--------------------
Cerchio con raggio 1.0
Area cerchio =  3.141592653589793
Circonferenza cerchio =  6.283185307179586
--------------------
Quadrato con lato 1.0
Area quadrato =  1.0
Perimetro quadrato =  4.0
--------------------
Rettangolo con lati 1.0 e  2.0
Area Rettangolo = 2.0
Perimetro Rettangolo = 6.0
```

Esempio completo in cui si usano in modo intercambiabile classi e funzioni

L'esempio della differenziazione

Voglio applicare la formula del rapporto incrementale in una dimensione ad una generica funzione f(t).

\frac{f(t+h)-f(t)}{h}

Per fare cio' definisco una nuova funzione differ che accetta come parametri:

- la funzione f da differenziare
- l'incremento h
- il valore della variabile indipendente

In [29]:
```
def differ( f , t , h=1e-6):
    return (f(t+h)-f(t))/h
```

la funzione diff accetta come parametri:

- la funzione f da differenziare
- il valore della variabile indipendente t presso cui effettuare la differenziazione
- l'incremento h

La funzione diff e' adatta per la differenziazione di una funzione di una sola variabile ad esempio:

In [30]:
```
def g(t):
    return t**4 +2*t**3
```

determino g e la sua derivata in t=1

In [31]:
```
g1 = g(1)
dg1= differ(g,1,h=0.001)
print(g1)
print(dg1)
```

```
3
10.012006000998941
```

Se ho una funzione che accetta piu' di un parametro non posso applicare direttamente la differenziazione ma ho diverse opzioni.

consideriamo la seguente:

In [32]:
```
from math import exp
def g(t,a,A):
    return A*exp(-a*t)
```

come fare ad applicare a g(t,a,A) la funzione differ (o qualcosa che sorta il medesimo effetto)?

1) replico le funzioni in base ai parametri

In [33]:
```
def g1(t):
    a=0.6
    A=2.
    return A*exp(-a*t)

def g2(t):
    a=2.
    A=1.7
    return A*exp(-a*t)
```

uso:

In [34]:
```
g1(1),differ(g1,1,h=0.001)
```

Out[34]: (1.097623272188053, -0.6583764306324902)

In [35]:
```
g2(1),differ(g2,1,h=0.001)
```

Out[35]: (0.2300699815022416, -0.4596801296480879)

ma se ho 100 set di variabili devo creare 100 funzioni!!!!!

2) variabili globali

dichiaro A ed a come variabili globali

In [36]:
```
def g(t):
    global a,A
    return A*exp(-a*t)
```

ogni volta che voglio cambiare i parametri modifico quelli globali

In [37]:
```
#imposto i parametri a livello globale
a=0.6
A=2.
#determino g e la sua derivata in t=1
g(1),differ(g,1)
```

Out[37]: (1.097623272188053, -0.6585737657438528)

In [38]:
```
#reimposto i parametri
a=2
A=1.7
#determino g e la sua derivata in t=1
g(1),differ(g,1)
```

Out[38]: (0.2300699815022416, -0.46013950283363414)

3) classe usata come una funzione

- trasformiamo i parametri a e A della funzione in attributi della classe
- creiamo una funzione membro che accetti come unico parametro la variabile indipendente t

In [39]:
```
class G:
    #inizializzo i parametri nel costruttore
    def __init__(self,a,A):
        self.a = a
        self.A = A

    def valore(self,t):
        return self.A*exp(-self.a*t)
```

uso:

In [40]:
```
#definisco la funzione e i suoi parametri
g1 = G( 0.6 , 2.0)
#determino il valore di g1 e della sua derivata in t=1
g1.valore(1),differ(g1.valore,1,h=0.001)
```

Out[40]: (1.097623272188053, -0.6583764306324902)

In [41]:
```
#definisco una nuova funzione con parametri diversi
g2 = G( 2.0 , 1.7)
#determino il valore di g1 e della sua derivata in t=1
g2.valore(1),differ(g2.valore,1,h=0.001)
```

Out[41]: (0.2300699815022416, -0.4596801296480879)

Possiamo far assomigliare del tutto le istanze g1 e g2 ad una funzione definendo il metodo standard __call__

In [42]:
```
class G:
    #inizializzo i parametri nel costruttore
    def __init__(self,a,A):
        self.a = a
        self.A = A

    #se associo una sola funzione alla classe
    #posso usare l'operatore __call__
    def __call__(self,t):
        return self.A*exp(-self.a*t)
```

uso:

In [43]:
```
#definisco la funzione e i suoi parametri, come sopra
g1 = G( 0.6 , 2.0)
#determino il valore di g1 e della sua derivata in t=1
#NB: uso g come se fosse una funzione
g1(1), differ(g1,1)
```

Out[43]: (1.097623272188053, -0.6585737657438528)

In [44]:
```
g2 = G( 2.0 , 1.7)
#definisco una nuova funzione con parametri diversi
```

```
g2(1), differ(g2,1)
```

Out[44]: (0.2300699815022416, -0.46013950283363414)

4) Soluzione basata su funzioni con numero variabile di argomenti (posizionali)

Non c'entra nulla con le classi ma e' opportuno citarla perche' valida e caratteristica del linguaggio.

cambio la funzione differ da:

In [45]:
```
def differ(f,t,h=1e-6):
    return (f(t+h)-f(t))/h
```

a:

In [46]:
```
def differ(f,t,h,*fpars):
    return ( f(t+h,*fpars) - f(t,*fpars) ) / h #NB: chiamo f(t,*fpars)
```

Inoltre considero la primissima implementazione della funzione:

In [47]:
```
def g(t,a,A):
    return A*exp(-a*t)
```

A questo punto la sintassi per determinare la derivata di g in 1 con a=0.6 e A=2.0 e' la seguente:

In [48]:
```
g(1,0.6,2.0),differ(g,1,0.001,0.6,2.0)
```

Out[48]: (1.097623272188053, -0.6583764306324902)

In [49]:
```
g(1,2.0,1.7),differ(g,1,0.001,2.0,1.7)
```

Out[49]: (0.2300699815022416, -0.4596801296480879)

5) anche differ diventa una classe

dove

- gli attributi della classe sono gli argomenti della funzione differ
- il metodo call implementa la logica di differ e gestisce eventuali parametri della funzione da differenziare

In [50]:
```
class Derivata:
    def __init__(self,f,h=1e-6):
        self.f = f
        self.h = h
    def __call__(self,t,*fpars):
        #NB: chiamo self.f(t,*fpars)
        return ( self.f(t+self.h,*fpars) - self.f(t,*fpars) ) / self.h
```

In [51]:
```
def g(t,a,A):
    return A*exp(-a*t)
```

la sintassi diventa estremamente pulita, infatti definisco una nuova funzione dg (che in realtà è una istanza della classe Derivata) che posso invocare con gli stessi argomenti della funzione di partenza

In [52]:
```
#dg è un oggetto di tipo Derivata che puo' essere invocato come una funzione
#rappresenta la derivata della funzione indipendentemente dai suoi parametri
dg = Derivata(g)
#i parametri sono specificati nel momento in cui effettuo il calcolo vero e proprio
g(1,0.6,2.0) , dg(1,0.6,2.0)
```

Out[52]: (1.097623272188053, -0.6585737657438528)

```
```

In [53]:
```
g(1,2.0,1.7) , dg(1,2.0,1.7)
```

Out[53]:
```
(0.2300699815022416, -0.46013950283363414)
```

6) con una chiusura

Anche questo c'entra il giusto con le classi ma è un'ottima alternativa e molto caratteristica.

In [54]:
```
def G(a,A):
    def g(t):
        return A*exp(-a*t)
    return g
```

In [55]:
```
g1 = G( 0.6 , 2.0)
g1(1),differ(g1,1,h=0.001)
```

Out[55]:
```
(1.097623272188053, -0.6583764306324902)
```

In [56]:
```
g2 = G( 2.0 , 1.7)
g2(1),differ(g2,1,h=0.001)
```

Out[56]:
```
(0.2300699815022416, -0.4596801296480879)
```

Esercizio

Data la seguente funzione che approssima l'integrale definito di una generica funzione f tra a e b con la formula dell'integrazione trapezioidale su n intervalli:

In [57]:
```
def trapezoid(f, ta, tb, n):
    """Approssima l'integrale definito di f tra a e b
con la formula dell'integrazione trapezioidale su n intervalli"""
    h = (tb - ta) / n
    s = f(ta) + f(tb)
    for i in range(1, n):
        s += 2 * f(ta + i * h)
    return s * h / 2
```

Ripetere il percorso fatto nei paragrafi precedenti per la differenziazione, applicandolo alla funzione parametrica

In [58]:
```
from math import exp
def g(t,a,A,B):
    return A*exp(-a*t) + B
```

In particolare sviluppare le seguenti 6 varianti, utilizzando come test l'integrazione tra 0 ed 1 e le due terne di parametri (a,A)

- (0.6,2.0,0.0) risultato integrale 1.504
- (2.0,1.7,0.7) risultato integrale 0.735

- trapezioid_v1.py: fissare i parametri in diverse versioni di g
- trapezioid_v2.py: usare parametri a ed A 'globali'
- trapezioid_v3.py: definire g come classe ed i parametri come attributi (2 versioni: usando e non usando l'operatore call)
- trapezioid_v4.py: ridefinire trapezioid in modo che operi su funzioni con numero variabile di argomenti (posizionali)
- trapezioid_v5.py: trasformare trapezioid in una classe TrapezIntegral
- trapezioid_v6.py: con una chiusura