Corso di **Python**

Lezione 14

Argomenti avanzati

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Cosa vedremo oggi

- Decoratori e chiusure
- Serializzazione
- Iteratori (cenni)

Gli esempi di questa lezione sono estratti in particolare dalle seguenti fonti:

- Esempi dal libro Fluent Python di Luciano Ramalho, pubblicati con licenza MIT sul sito https://github.com/fluentpython/example-code
- Python Module of the Week:
  - https://pymotw.com/2/contents.html
  - https://pymotw.com/3/ (versione per Python 3)

Decoratori

- Abbiamo già visto che in linea generale in Python i decoratori sono funzioni che hanno come argomento una funzione:

```
def some_function():
    # function body…
some_function = some_decorator(some_function)
```

  che più spesso si usano con la sintassi speciale:

```
@some_decorator
def some_function():
    # function body…
```

- La sintassi dei decoratori è definita nel PEP 318:
  - https://www.python.org/dev/peps/pep-0318/
- Questa definizione è estensiva rispetto a quella OOP (*Design Pattern* della cosiddetta *Gang of Four*)

Funzioni come oggetti

- In Python le funzioni sono first-class object, ossia possono essere passate come argomenti ad altre funzioni
- La funzione ritornata dal decoratore generalmente non corrisponde a quella originale:

```
>>> def decorate(func):
...     def inner():
...         print("Running inner ...")
...     return inner
...
>>> @decorate
... def target():
...     print("Running target ...")
...
>>> target()
Running inner ...
>>> target
<function decorate.<locals>.inner at 0x7f1edb2c6a60>
```

Esecuzione dei decoratori

- I decoratori sono eseguiti quando il modulo viene caricato
- Avendo ad esempio un decoratore che rigistra le funzioni in un elenco (sì, questo decoratore non modifica la funzione!):

```
registry = []

def register(func):
    print('running register(%s)' % func)
    registry.append(func)
    return func
```

quando lo useremo su una funzione questo verrà immediatamente applicato:

```
@register
def f1():
    print('running f1()')

def f2():
    print('running f2()')
```

Scope delle variabili

- Abbiamo già visto che ci sono variabili globali e variabili locali:

```
b = 10
def f1(a):
    print(a)
    print(b)     # Considerata globale
```

- Se nella funzione riassegno *b* allora viene considerata implicitamente locale:

```
b = 10
def f1(a):
    print(a)
    print(b)     # Ora b è considerata locale --> Errore runtime
    b = 11
```

- In questo caso devo esplicitare che si tratta di una variabile globale:

```
b = 10
def f1(a):
    global b
    ...
```

Disassemblatore

- Per verificare il diverso comportamento nello scope potete far uso del disassemblatore di byte code:

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> from dis import dis
>>> dis(f1)
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_FAST                0 (a)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_GLOBAL              0 (print)
             13 LOAD_GLOBAL              1 (b)
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP
             20 LOAD_CONST               0 (None)
             23 RETURN_VALUE
```

https://docs.python.org/2/library/dis.html

Closures - Chiusure

- Una closure è una funzione che utilizza uno scope esteso a variabili non locali
- Esempio:
  - Calcolatore di medie per serie di numeri generati nel tempo:

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

- Vediamo 2 modi diversi per realizzarlo:
  - Metodo OOP
  - Funzionale con *closure*

Media con metodo OOP

- Con le classi si potrebbe realizzare un memorizzatore di valori medi medie usando una lista come attributo e il metodo speciale `__call__`:

```
class Averager():
    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
```

Metodo con closure

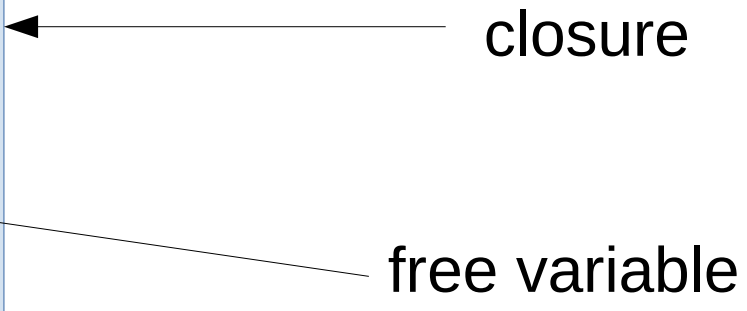
- Il metodo funzionale prevede una funzione che restituisce una funzione che calcola una media:

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)    # Estensione scope a variabile non locale
        total = sum(series)
        return total/len(series)

    return averager
```

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
```

Cos'è quindi una closure?

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```



- All'interno di averager la variabile series non è legata allo scope locale (free variable)
- Una closure in sostanza è una funzione che conserva il legame con le variabili non locali che esistono quendo la funzione viene definita
  - Queste variabili non locali della closure continano ad esistere anche dopo che lo scope in cui la funzione è stata definita non esiste più

Statement **nonlocal**

- Ipotizzando di voler realizzare una versione più efficiente che computa la media in modo progressivo:

```
def make_averager()
    count = 0
    total = 0
    def averager(new_value):
        count += 1              # crea un nuovo count locale!
        total += new_value      # crea un nuovo total locale!
        return total / count
    return averager
```

- Questa closure non funziona perché usa oggetti immutabili: questi vengono ricreati all'interno della funzione interna e pertanto sono interpretati come variabili locali
- In questi casi in Python 3 è possibile inserire uno statement ***nonlocal*** all'inizio della funzione interna:

```
def averager(new_value):
    nonlocal count, total
    ...
```

- In Python 2 questo statement non c'è e la soluzione di ripiego è quella di inserire gli oggetti immutabili all'interno di contenitori mutabili (liste o dizionari)

Esempio di decoratore 1/2

- Un decoratore che aggiunge un contatore di tempo e visualizza tempo trascorso, argomenti passati e risultato delle chiamate:

```
import time


def clock(func):
    def clocked(*args):
        t0 = time.time()
        result = func(*args)
        elapsed = time.time() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked
```

https://raw.githubusercontent.com/fluentpython/example-code/master/07-closure-deco/clockdeco.py

Esempio di decoratore 2/2

- Decoriamo un funzione che calcola il fattoriale con il decoratore *clock*:

```
>>> @clock
... def factorial(n):
...     return 1 if n < 2 else n * factorial(n-1)
...
>>> factorial(5)
[0.00000286s] factorial(1) -> 1
[0.00010800s] factorial(2) -> 2
[0.00016356s] factorial(3) -> 6
[0.00022030s] factorial(4) -> 24
[0.00029063s] factorial(5) -> 120
```

- In questa versione minimale di decoratore il nome della funzione originale viene mascherato:

```
>>> factorial.__name__
'clocked'
```

Decorare metodi di una classe

- Per decorare metodi di una classe si devono utilizzare decoratori che includono il parametro *self*:

```
def p_decorate(func):
    def func_wrapper(self):
        return "<p>{0}</p>".format(func(self))
    return func_wrapper

class Person(object):
    def __init__(self, name, family):
        self.name = name
        self.family = family

    @p_decorate
    def get_fullname(self):
        return '%s %s' % (self.name, self.family)

>>> my_person = Person("Davide", "Brunato")
>>> print(my_person.get_fullname())
```

Decoratori built-in per classi

`property`: Decoratore applicabile a metodi che includono solo il parametro di istanza *self*:

```
@property
def weight(self):
    return self.__weight
```

`classmethod`, **`staticmethod`**: Decoratori che modificano un metodo in modo che non riceva il parametro relativo all'istanza; classmethod include anche un primo parametro relativo alla classe

```
class Demo:
    @classmethod
    def myclassmeth(*args):
        return args
   @staticmethod
   def mystaticmeth(*args):
        return args


>>> Demo.myclassmeth('test')
(<class '__main__.Demo', 'test')
>>> Deomo.mystaticmeth('test')
('test',)
```

Includere gli argomenti

- Per definire decoratori applicabili a funzioni con diversi e differenti argomenti usare la forma generica con argomenti variabili, posizionali e con *keywords*:

```
from time import sleep


def sleep_decorator(function):
    """Limita la velocità con cui una funzione può essere chiamata"""

    def wrapper(*args, **kwargs):
        sleep(2)
        return function(*args, **kwargs)
    return wrapper

@sleep_decorator
def print_number(num):
    return num
```

Libreria **functools**

- Include decoratori di alto livello, ad esempio utili per il problema del mascheramento del nome originale:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Decoratore applicato alla funzione di decorazione!

Decoratore **lru_cache**

- Il decoratore `functools.lru_cache` può essere molto utile qualora si vogliano salvare i risultati di precedenti esecuzioni
- Esempio:

```
@functools.lru_cache()
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Decorazione multipla

- Si possono applicare più decoratori impilandoli:

```
@d1
@d2
def f():
    print('f')
```

equivale alla seguente:

```
def f():
    print('f')

f = d1(d2(f))
```

Esempio di decorazione multipla

```
import functools
from clockdeco import clock

@functools.lru_cache()
@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__=='__main__':
    import sys
    print(fibonacci(int(sys.argv[1])))
```

- Provando includendo o escludendo il decoratore lru_cache si verificano le differenze:

```
$ time python3 fibo_demo_lru.py 20 > /dev/null

real  0m1.739s
user  0m0.113s
sys   0m0.035s
$ time python3 fibo_demo_lru.py 20 > /dev/null

real  0m0.024s
user  0m0.021s
sys   0m0.004s
```

Decoratori con argomenti

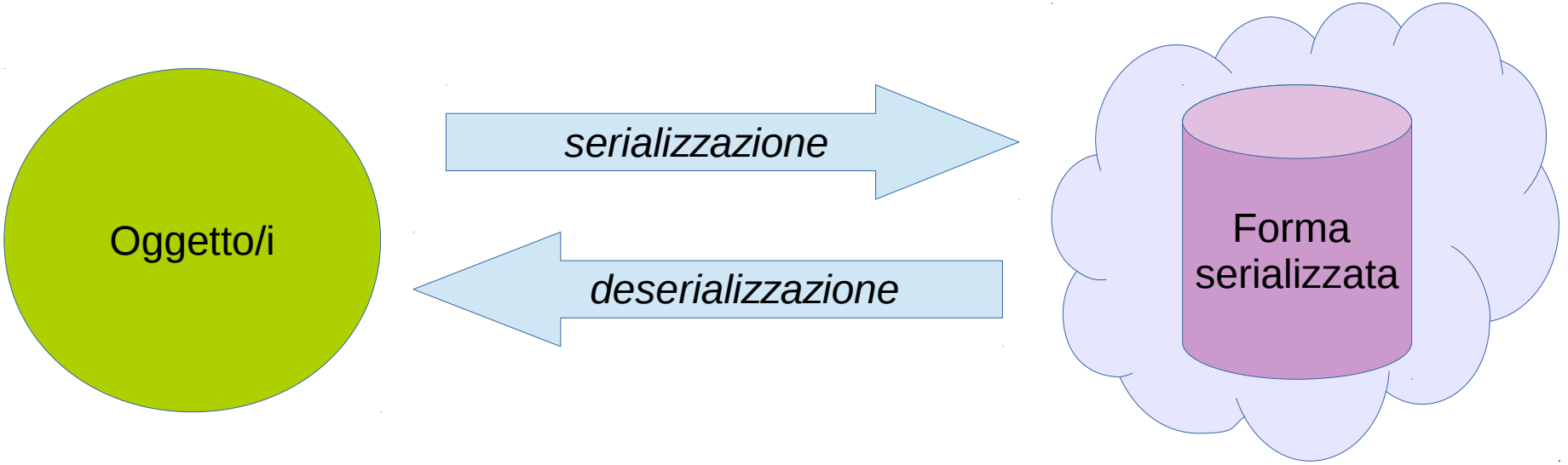
- Il decoratore può includere dei parametri:

```
registry = set()

def register(active=True):
    def decorate(func):
        print('running register(active=%s)->decorate(%s)' % (active, func))
        if active:
            registry.add(func)
        else:
            registry.discard(func)
        return func
    return decorate

@register(active=False)
def f1():
    print('running f1()')
```

- I parametri devono essere definiti al momento dell'applicazione del decoratore
- Il decoratore diventa un *decorator factory*, molto simile al caso di decoratore in stack

Serializzazione

- Processo per salvare un oggetto in un supporto di memorizzazione o per trasmetterlo su una connessione di rete
- Il formato può essere in binario o testuale (es. XML)
- Scopo della serializzazione è salvare o trasmettere l'intero stato dell'oggetto, in modo che possa essere successivamente ricreato in modo identico con un processo detto di *deserializzazione*



Il modulo **pickle**

- Implementa un algoritmo per trasformare oggetti Python in sequenza di bytes
- Pickle implementa una serializzazione di tipo binario, con un formato dei dati è specifico di Python
- È implementato sia in puro Python che in linguaggio C
  - In Python 2 sdoppiamento esplicito in 2 librerie `pickle` e `cPickle`
  - In Python 3 l'import è unico (`pickle`) ma è l'implementazione che si incarica di provare il caricamento della versione compilata

```
# Import compatibile Python 2.x/3.x
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

- La versione in puro Python è utile in quanto con cPickle non si possono fare sottoclassi
- Pickle serializza oggetti, non la definizione delle classi:
  - La ricostruzione richiede che la classe sia nello spazio dei nomi del codice chiamante
  - Il protocollo gestisce i riferimenti circolari tra gli oggetti

Formati di pickle

- Definite 5 versioni diverse di protocollo, per renderlo più efficienti e adeguarlo all'evoluzione del linguaggio:

  1) Protocollo **versione 0**: formato originale testuale, compatibile con le prime versioni del linguaggio Python.

  2) Protocollo **versione 1**: vecchio formato binario compatibile con le prime versioni di Python.

  3) Protocollo **versione 2**: introdotto in Python 2.3 (vedere PEP 307), che fornisce una più efficiente serializzazione delle classi new-style.

  4) Protocollo **versione 3**: aggiunto in Python 3.0 per supportare il tipo bytes. Non può essere deserializzato da Python 2.x.

  5) Protocollo **versione 4**: aggiunto in Python 3.4 (PEP 3154). Aggiunge il supporto alla serializzazione di oggetti molto grandi e di più tipologie di oggetti e alcune ottimizzazioni del formato.

- La versione 3 del protocollo è quella di default e anche quella consigliata quando è richiesta compatibilità con le varie versioni di Python 3

Esempio base d'uso

- Esempio di serializzazione di una struttura dati con Python 3.4:

```
>>> import pickle
>>> import pprint
>>> data = [{'a':'A', 'b':2, 'c':3.0}]
>>> pprint.pprint(data)
[{'a': 'A', 'b': 2, 'c': 3.0}]
>>> data_string = pickle.dumps(data)
>>> print(data_string)
b'\x80\x03]q\x00}q\x01(X\x01\x00\x00\x00bq\x02K\x02X\x01\x00\
x00\x00aq\x03X\x01\x00\x00\x00Aq\x04X\x01\x00\x00\x00cq\x05G@
\x08\x00\x00\x00\x00\x00\x00ua.'
```

- Il formato usato è diverso da quello di Python 2.7 che usa la versione 2 del protocollo

Pickling e Unpickling

- Per codificare si usa la funzione `pickle.dumps`, mentre la decodifica si opera con la funzione `pickle.loads`:

```
>>> data1 = [{'a':'A', 'b':2, 'c':3.0}]
>>> data_string = pickle.dumps(data1)
>>> data2 = pickle.loads(data_string)
>>> data2
[{'a': 'A', 'c': 3.0, 'b': 2}]
>>> data1 is data2   # Sono oggetti distinti ...
False
>>> data1 == data2   # ... ma sono effettivamente identici
True
```

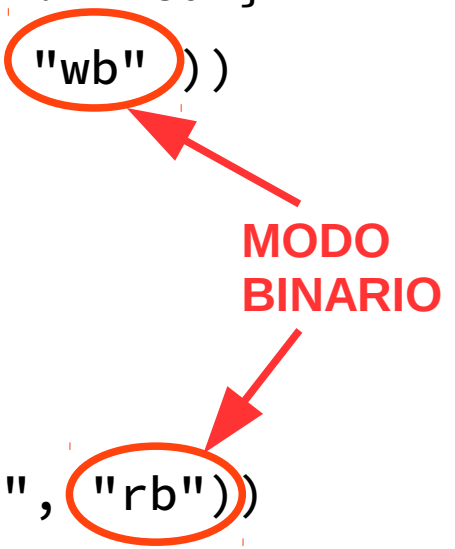
Pickling su file

- Per serializzare direttamente su file c'è la funzione **`pickle.dump`** (senza la *esse* ...):

```
>>> import pickle
>>> favorite_color = {"lion": "yellow", "kitty": "red"}
>>> pickle.dump(favorite_color, open("save.p", "wb"))
```

- Per deserializzare in analogia c'è la funzione **`pickle.load`**:

```
>>> import pickle
>>> favorite_color2 = pickle.load(open("save.p", "rb"))
>>> favorite_color2
{'kitty': 'red', 'lion': 'yellow'}
```



Serializzare un oggetto

- La serializzazione può operare su oggetti generici:

```
import pickle
from io import BytesIO
class SimpleObject(object):
    def __init__(self, name):
        self.name = name
        l = list(name)
        l.reverse()
        self.name_backwards = ''.join(l)
        return

obj = SimpleObject('last')
out_stream = BytesIO()        # StringIO se si usa Python 2.x
pickle.dump(obj, out_stream)
out_stream.flush()
in_stream = BytesIO(out_stream.getvalue())
obj = pickle.load(in_stream)
print('READ: %s (%s)' % (obj.name, obj.name_backwards))
```

Altri moduli Python per serializzazione interna

- **marshal**: serializzazione con formato binario Python tipo pseudo-code *.pyc*
  - https://docs.python.org/3/library/marshal.html
  - Non è un modulo orientato alla persistenza dell'oggetto, in quando il formato dello pseudo code può variare da versione a versione

- **shelve**: Utilizza i moduli pickle e dbm per implementare la persistenza su strutture di tipo dizionario (*shelf-scaffale*):
  - https://docs.python.org/3/library/shelve.html
  - Le strutture sono lette su file scritti con *dbm.dumb*:

```
import shelve
s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```

Serializzare con JSON

- JSON è un formato di serializzazione di tipo testo:
  - http://json.org/
  - Prodotto come testo Unicode, generalmente codificato in utf-8
- JSON un formato leggibile più di altri standard (es. XML)
- JSON è interoperabile con l'esterno
- JSON codifica solamente un sottoinsieme dei tipi built-in di Python
- Esempio di codifica in JSON:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Mappatura Python - JSON

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

JSON: Esempio d'uso

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

JSON: limitazione sulle chiavi

- Le chiavi dei dizionari devono essere stringhe, è possibile però escludere quelle non conformi:

```
import json

data = [ { 'a':'A', 'b':(2, 4), 'c':3.0, ('d',):'D tuple' } ]

print('First attempt')
try:
    print(json.dumps(data))
except (TypeError, ValueError) as err:
    print('ERROR:', err)

print("")
print('Second attempt')
print(json.dumps(data, skipkeys=True))
```

Iterabili e iteratori

- Gli **iterabili** sono oggetti i cui elementi possono essere elencati secondo un ordine più o meno prestabilito
  - Le strutture dati contenitori (tuple, liste, insiemi, dizionari) sono iterabili
  - I file o gli stream sono iterabili, eventualmente senza un limite predefinito
- Sugli oggetti iterabili possono essere creati degli **iteratori**
  - Per creare l'iteratore si usa la funzione built-in `iter()`
  - Gli iteratori sono oggetti sui usati nei loop per percorrere l'oggetto iterabile
  - Vengono usati in modo automatico dal ciclo *for* o in alternativa si possono gestire manualmente utilizzando la funzione built-in `next()`
  - Quando gli elementi sono terminati viene generata l'eccezione `StopIteration`
  - Gli iteratori permettono di risparmiare memoria evitando duplicazioni dell'oggetto da iterare

Esempio di iteratore

```
>>> a = [ 'a' , 'b', 'c']
>>> x = iter(a)
>>> x.next()
'a'
>>> x.next()
'b'
>>> x.next()
'c'
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Definire iteratori

- Per definire iteratori complessi c'è la libreria `itertools`
- Un esempio di funzione disponibile in questa libreria:

```
itertools.ifilter(predicate, iterable)
```

  - Crea un iteratore che filtra gli elementi dell'iterabile, includendo solo quelli per i quali il predicato è True. Se il predicato è None allora ritorna solo gli elementi che sono True. Equivale alla seguente funzione:

```
def ifilter(predicate, iterable):
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

- Esempio:

```
>>> from itertools import ifilter
>>> x = ifilter(lambda x: x%2, range(10))
>>> [ i for i in x)   # Equivale a list(x)
[1, 3, 5, 7, 9]
>>> [ i for i in x]    # Iteratore esaurito!! Usare itertools.tee per duplicarlo ...
[]
```