Corso Python 2018

POLITECNICO OPEN
unix LABS
Come hack with us.

- Giacomo Vercesi
  1vercesig_at_gmail.com
- Edoardo Negri edne_at_gmx.com

Programmazione ad Oggetti

Potete ottenere questa presentazione recandovi all'indirizzo:

slides.poul.org/2018/python

Perché la programmazione ad oggetti?

- perché in Python tutto è un oggetto
- permette di modellare il programma come un insieme di entità comunicanti tra di loro
- racchiudere insieme le variabili e il codice che le modifica

Cos’è una classe

- un contenitore di variabili e funzioni
- le funzioni modificano queste variabili
- le variabili si chiamano **attributi**
- le funzioni si chiamano **metodi**

Cos'è un'oggetto

È quando una classe prende vita.

```
class Classe:
    pass

oggetto = Classe()
```

Si dice che `oggetto` è una **istanza** di `Classe`.

Scavare dentro gli oggetti

Il comando `dir()` restituisce il contenuto di un oggetto, attributi e metodi.

```
>>> dir("Ciao")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
...
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

- La variabile `self`
  - Il primo parametro di ogni metodo è sempre l'istanza su cui si sta operando
  - Convenzionalmente si chiama `self`
- La funzione `__init__()`:
  - Inizializza l'oggetto

```
class Contatore():
    def __init__(self):
        self.numero = 0

    def aumenta(self, incremento=1):
        self.numero += incremento
```

Perché tutti questi *underscore*?

È una convenzione.

- **`variabile`**: variabile generica
- **`_variabile`**: variabile privata. È consigliato non modificarla
- **`__variabile`**: variabile interna. Modificarla potrebbe creare problemi
- **`__variabile__`**: variabile interna del linguaggio. Da non modificare a meno di non sapere cosa si sta facendo
- **`variabile_`**: da usare se esiste una variabile built-in con lo stesso nome

Duck Typing

«If it looks like a duck, walks like a duck and quacks like a duck then it just may be a duck» ***—Walter Reuther***

«Se un oggetto ha un metodo `read()`*,* `write()` *e* `seek()`*,* `flush()`*, ... allora è un file»* ***—Python***

`__getitem__()`

- Duck typing: se ha un metodo `__getitem__` allora “si comporta come” una lista
- Possiamo accedere ai suoi elementi usando la sintassi `oggetto[indice]`

```
class Tabellina8():
    def __getitem__(self, i):
        return 8*i
```

```
>>> t = Tabellina8()
>>> t[3]
24
```

Alcuni esempi aggiuntivi

- `'*'` → `__mul__(a, b)`
- `'+'` → `__add__(a, b)`
- `print` → `__str__()`

Per una lista più esaustiva (https://docs.python.org/3/reference/datamodel.html)

Ereditarietà

- Permette di definire sottoclassi (sottoinsiemi di oggetti).
- La classe figlio ottiene attributi e metodi dalla classe padre e li può ridefinire o aggiungerne altri.

```python
class Contatore():
    def __init__(self):
        self.numero = 0

    def aumenta(self, incremento=1):
        self.numero += incremento


class ContatoreStampabile(Contatore):
    def __repr__(self):
        return str(self.numero)
```

`super()`

La funzione `super()` permette di accedere ad attributi e medodi della classe padre.

Eccezioni

Cosa sono?

Le eccezioni sono un meccanismo per comunicare che c’è una situazione anomala all’interno del programma

Come gestiamo le eccezioni?

```
a = int(input("Inserisci numeratore: "))
b = int(input("Inserisci denominatore: "))

try:
    c = a/b
    print("{}/{} è {}".format(a,b,c))
except ZeroDivisionError:
    print("Denominatore 0, impossibile eseguire divisione)
```

Sollevare un’eccezione

Usiamo il costrutto ‘raise()’.

```
raise ZeroDivisionError("Messaggio di errore")
```

Definire le eccezioni

```
class IncrementoZeroError(Exception):
    pass
```

Getter e Setter

Getter

```
class Vettore:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @property
    def modulo(self):
        return (self.x**2 + self.y**2)**0.5

>>> v = Vettore(3, 4)
>>> v.modulo
5.0
```

Setter

```
class Vettore:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @property
    def modulo(self):
        return (self.x**2 + self.y**2)**0.5

    @modulo.setter
    def modulo(self, m):
        self.x = m * self.x/self.modulo
        self.y = m * self.y/self.modulo

>>> v = Vettore(3, 4)
>>> v.modulo = 25
```

Seconda Parte

`if` ternario

```
z = x if condizione else y
```

Comprehension

```
lista = [f(x) for x in collezione if condizone(x)]
```

- Con le parentesi graffe si definisce un `set`
- Con le tonde un generatore (ne parleremo più avanti)
- Si possono fare anche dizionari

```
dizionario = {chiave: valore for ...}
```

Esempi:

```
# lista dei file png nella cartella corrente
[f for f in os.listdir('.') if f.endswith('.png')]

# {nome_file: last_acces_time}
{f: os.path.getatime(f) for f in os.listdir('.')}
```

Nota sull'indentazione

- Aperta parentesi e a capo

```
l = [
  f for f in os.listdir('.')
  if f.endswith('.png')
]
```

- Oppure allineato alla parentesi

```
l = [f for f in os.listdir('.')
     if f.endswith('.png')]
```

`any` e `all`

```
>>> any([False, False, True])
True

>>> all([True, False])
False
```

`*args` e `**kwargs`

Ovvero gli asterischi

*args

Posso “spacchettare” una lista (o tupla) ed usarne i valori come argomenti di una funzione

```
def f(a, b):
    return a + b

>>> l = [1, 2]
>>> f(l[0], l[1])
3
>>> f(*l)
3
```

```
def g(a, b, c):
    return a+b+c

>>> g(3, *l)
6
```

Oppure “impacchettare” qualsiasi numero di argomenti dentro una tupla:

```
def f(*args):
    return len(args)

>>> f('a', 'b', 'c')
3
```

```
>>> def g(a, *args):
...     print(a, args)

>>> g(1, 2, 3)
1 (2, 3)
```

`**kwargs`

Simile ma con dizionari

- Chiavi: nomi degli argomenti

```
def f(a, b):
    print(a, b)

>>> d = {'a': 1, 'b': 2}
>>> f(**d)
1 2
```

Esempio

```
>>> def f(a, b, *args, **kwargs):
...     print(a, b)
...     print(args)
...     print(kwargs)

>>> f(1, 2, 3, 4, c=5, xyz=6)
1 2
(3, 4)
{'xyz': 6, 'c': 5}
```

Esempio

```
def f(*args, **kwargs):
    print(args, kwargs)

>>> f(1, 2, 3, a=12, b='asd')
(1, 2, 3) {'a': 12, 'b': 'asd'}
```

Chiusure

Funzioni dentro funzioni

Se una funzione viene ritornata ha ancora accesso alle variabili della funzione più esterna

```
def new_greeter(name):
    def greet():
        print("Hello,", name)

    return greet


greet_jack = new_greeter("Jack")
greet_jack()  # Hello, Jack
```

Decoratori

Decoratori: utilizzo

```python
@app.route('/api/queue/current')
def now_playing():
    uuid = transmitter_queue.now_playing
    track = get_tags(uuid) if uuid else {}

    return jsonify(track=track), 200
```

(fonte)

Decoratori: come definirli

```
def chiedi_conferma(f):
    def g():
        if input('Sicuro? [y, n] ') == 'y':
            f()
    return g


@chiedi_conferma
def funzione():
    print('done')


>>> funzione()
Sicuro? [y, n] y
done
```

Decoratori con argomenti

funzione che restituisce un decoratore

```
def chiedi_conferma(prompt='Sicuro?'):
    def decoratore(f):
        def g():
            if input(prompt + ' [y, n] ') == 'y':
                f()
        return g
    return decoratore


@chiedi_conferma('Sei veramente sicuro?')
def funzione():
    print('done')


>>> funzione()
Sei veramente sicuro? [y, n] y
```

Generatori

- Lazy evaluation
- Risparmiare memoria e computazione

yield transorma una funzione in un generatore

```
def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a+b
```

`next()`

Vengono “consumati”

Thank you!

POLITECNICO OPEN
unix LABS
Come hack with us.