OOP: Object Oriented Programming

Riferimento: Capitolo 8 del testo di Python consigliato
"Classes and object oriented programming"

Documentazione online:
https://docs.python.org/3.6/tutorial/classes.html

OOP: Idea Principale

L'idea principale della programmazione orientate agli oggetti è di pensare agli **oggetti** come delle collezioni sia di **dati** che di **metodi** che operano su quei dati.

Gli oggetti sono una caratteristica dominante di Python: ogni **oggetto** ha un **tipo** che definisce il tipo di **operazioni** che un programma può eseguire su quell'oggetto.

Abbiamo visto come definire nuove **funzioni**, ora vediamo come definire nuovi **tipi**

Esempi: list, string, dictionary, set, …

Liste

https://docs.python.org/3/tutorial/datastructures.html

5.1. More on Lists ¶

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)
Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`clear`**()
Remove all items from the list. Equivalent to `del a[:]`.

`list.`**`index`**(*x*[, *start*[, *end*]])
Return zero-based index in the list of the first item whose value is *x*. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

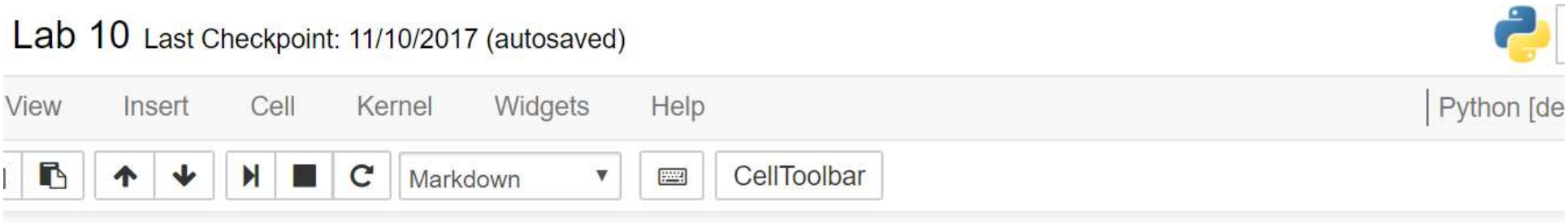
`list.`**`count`**(*x*)
Return the number of times *x* appears in the list.

`list.`**`sort`**(*key=None*, *reverse=False*)
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).

`list.`**`reverse`**()
Reverse the elements of the list in place.

`list.`**`copy`**()
Return a shallow copy of the list. Equivalent to `a[:]`.

Stringhe



Funzioni builtin per le stringhe

I seguenti metodi sono tutti molto utili e restituiscono delle nuove stringhe lasciando la stringa iniziale immutata:

- `s.count(s1)`: conta qualche volte la stringa `s1` è contenuta in `s`
- `s.find(s1)`: restituisce l'indice della stringa `s` in cui ha trovato per la prima volta la stringa `s1`; altrimenti restituisce -1
- `s.rfind(s1)`: come sopra, ma inizia dalla fine di `s` (la `r` sta per `reversed`)
- `s.lower()`: converte tutte le lettere in minuscolo
- `s.upper()`: converte tutte le lettere in maiuscolo
- `s.replace(old,new)`: sostituisce tutte le sotto stringhe uguali a `old` in `s` con la stringa `new`
- `s.strip()`: rimuove tutti i caratteri blanks iniziali e finali dalla stringa `s`
- `s.strip()`: rimuove tutti i caratteri blanks finali dalla stringa `s`
- `s.split(d)`: suddivide la stringa in sotto stringhe usando il carattere `d` come separatore

Dizionari

Lab 10 Last Checkpoint: 11/10/2017 (autosaved)

View Insert Cell Kernel Widgets Help

Markdown CellToolbar

Metodi utili sui dizionari

I seguenti metodi sono molto utili per usare i dizionari:

- `len(d)`: restituisce il numero di elementi nel dizionario `d`
- `d.keys()`: restituisce una lista (vista) delle chiavi del dizionario `d`
- `d.values()`: restituisce una lista (vista) dei valori del dizionario `d`
- `key in d`: restituisce `True` se la chiave `key` è nel dizionario `d`
- `d.get(key, value)`: restituisce `d[key]` se `key` è in `d`, altrimenti restituisce il valore `value`
- `d[key] = value`: associa il valore `value` alla chiave `key` nel dizionario `d`
- `del d[key]`: rimuove la chiave `k` dal dizionario `d`
- `for key in d`: itera sulle chiavi del dizionario `d`

Esempio: Numeri Complessi

Tipo: Numero complesso

Dati: due numeri reali, la parte reale e quella immaginaria

Operazioni: somma, stampa a video, calcolo del coniugato

La specifica delle operazioni possibili definisce una **interfaccia** tra i dati ed il resto del programma.

Il modo in cui viene calcolato il coniugato di un numero complesso (l'algoritmo usato e la sua implementazione) viene nascosto al resto del programma (**ENCAPSULATION**)

Abstract Data Types

Un **Abstract Data Type (ADT)** è un insieme di oggetti e di operazioni sugli oggetti stessi

Operazioni ed oggetti sono raggruppati in modo che possano essere passati da una parte all'altra del programma

Due concetti chiave della programmazione sono:

1. **DECOMPOSITION**: viene usata per creare una struttura nel nostro programma
2. **ABSTRACTION**: si cerca di eliminare i dettagli insignificanti per concentrarsi sugli aspetti fondamentali di un problema

ESEMPIO: *pensare sempre ai numeri complessi, quando vengono passati ad una funzione*

Classi

Un Abstract Data Type (ADT) è un insieme di oggetti e di operazioni sugli stessi oggetti. In Python, si implementano nuovi ADT definendo nuove classi

```
class NumeroComplesso:
    def __init__(self, real, imag):
        """ Metodo costruttore, chiamato quando viene
            inizializzato un nuovo oggetto """
        self.a = real
        self.b = imag

    def somma(self, c):
        """ Somma al numero corrente il numero complesso c """
        self.a = self.a + c.a
        self.b = self.b + c.b

    def __str__(self):
        """ Ritorna una stringa che rappresenta il numero """
        return str(self.a) + ' + ' + str(self.b) +'i'
```

Operazioni su Classi

Una classe supporta due tipi di operazioni:

1. **Istanziazione**: viene usata per creare una nuova **istanza** della classe, ovvero un nuovo oggetto del tipo definite dalla classe stessa. Quando viene creato un nuovo oggetto viene sempre richiamato il suo **metodo costruttore: `__init__`**
2. **Riferimento** ai suoi attribute **(metodo selettori)** : si usa la "**dot notation**" per acceddere agli attributi e metodi della classe

ATTENZIONE: l'oggetto associato all'espressione che precede il 'dot' viene implicitamente passato come primo parametro del metodo, e viene chiamato, per convenzione, sempre **`self`** (Vedi notebook)

Metodi con underscore __XX__

In Python esistono diversi metodi che possono essere definiti con un doppio underscore prima e dopo il nome del metodo, tipo `__init__`

ALTRI ESEMPI:

- `__str__(self)`: restiruisce una stringa, e viene per esempio chiamato in automatic quando un oggetto viene passatto alla funzione `print()`
- `__add__(self, other)`: viene utilizzato per fare l'OVERLOADING dell'operatore di addizione '+'
- `__eq__(self, other)`: viene utilizzato per fare l'OVERLOADING dell'operatore di confronto '=='

Metodi con underscore __XX__

You Want...	So You Write...	And Python Calls...
addition	`x + y`	`y.__radd__(x)`
subtraction	`x - y`	`y.__rsub__(x)`
multiplication	`x * y`	`y.__rmul__(x)`
division	`x / y`	`y.__rtruediv__(x)`
floor division	`x // y`	`y.__rfloordiv__(x)`
modulo (remainder)	`x % y`	`y.__rmod__(x)`
floor division & modulo	`divmod(x, y)`	`y.__rdivmod__(x)`
raise to power	`x ** y`	`y.__rpow__(x)`
left bit-shift	`x << y`	`y.__rlshift__(x)`
right bit-shift	`x >> y`	`y.__rrshift__(x)`
bitwise and	`x & y`	`y.__rand__(x)`
bitwise xor	`x ^ y`	`y.__rxor__(x)`
bitwise or	`x | y`	`y.__ror__(x)`

Metodi con underscore __XX__

You Want...	So You Write...	And Python Calls...
equality	`x == y`	`x.__eq__(y)`
inequality	`x != y`	`x.__ne__(y)`
less than	`x < y`	`x.__lt__(y)`
less than or equal to	`x <= y`	`x.__le__(y)`
greater than	`x > y`	`x.__gt__(y)`
greater than or equal to	`x >= y`	`x.__ge__(y)`
truth value in a boolean context	`if x:`	`x.__bool__()`

Documentazione online con tutti i possibili metodi di overloading:
http://www.diveintopython3.net/special-method-names.html

Inheritance

L'EREDITARIETÀ offre un meccanismo per costruire gruppi di tipi (classi) collegati tra loro attraverso una struttura gerararchica

In pratica permette di costruire una **gerarchia di tipi**, in cui un tipo di dati (**subclass**) può ereditare tutti gli attributi e metodi dal tipo da cui deriva (la sua **superclass**)

Si veda il notebook Lab 14 per un semplice esempio con i numeri complessi, per maggiori dettagli si rimanda al Capitolo 8 del libro di riferimento.

Object Oriented vs Functional Programming

L’uso di ADT incoraggia l’analista programmatore a pensare più in termini di **OGGETTI** piuttosto che di **FUNZIONI**

Un programma diventa una collezione di **TIPI** invece che una collezione di **FUNZIONI**

Esempio 1: Adder

- Una **CLASSE** rappresenta dei "*dati con delle operazioni collegate*"

- Una **CLOSURE** rappresenta delle "*operazioni con dei dati collegati*"

ESEMPIO: vedi notebook Lab 14

Esempio 2: Counter

- Una **classe** rappresenta dei "*dati con delle operazioni collegate*"

- Una **closure** rappresenta delle "*operazioni con dei dati collegati*"

ESEMPIO: vedi notebook

PARTE SECONDA:
HANDLING EXCEPTIONS

Handling Exceptions 1/4

Sino ad ora abbiamo visto le Exceptions come a degli errori gravi che causano un'interruzione "brutale" del nostro programma. Esempi visti a lezione:

- **ValueError**: invalid literal for int() with base 10: '3.0'
- **AttributeError**: readonly attribute (`z.real = 3`)
- **NameError**: name 'x' is not defined (dopo `del x`)
- **TypeError**: unsupported operand type(s) for ** or pow(): 'str' and 'int'
- **TypeError**: 'int' object is not iterable
- **StopIteration**: chiamata a next() su un iteratore vuoto

Handling Exceptions 2/4

Questi sono tutti esempi di eccezioni generate dal sistema per gestire ERRORI in parte prevedibili del programma.

- **ValueError**: invalid literal for int() with base 10: '3.0'
- **AttributeError**: readonly attribute (`z.real = 3`)
- **NameError**: name 'x' is not defined (dopo `del x`)
- **TypeError**: unsupported operand type(s) for ** or pow(): 'str' and 'int‘
- **TypeError**: 'int' object is not iterable
- **StopIteration**: chiamata a next() su un iteratore vuoto

Handling Exceptions 3/4

Gli errori precedenti sono generate dal commando

```
raise ValueError('Vostro msg errore in dialetto')
```

E possono essere gestiti tramite i comandi

```
try:
        ChiamataFunzione()
except ValueError:
        print('decidete voi cosa fare, ma no crash!')
```

Documentazione con tutte le possibili eccezioni:
https://docs.python.org/3/library/exceptions.html

Handling Exceptions 4/4

Questi sono tutti esempi di eccezioni generate dal sistema per gestire ERRORI in parte prevedibili del programma. Gli errori precedenti sono generate dal comando

```
raise ValueError('Vostro msg errore')
```

E possono essere gestiti tramite i comandi

```
try:
        ChiamataFunzione()
except ValueError:
        print('decidete voi cosa fare, ma no crash!')
```

LE ECCEZIONI SONO UN MODO PER SEMPLIFICARE IL FLUSSO DEL PROGRAMMA *(LO SONO??)*

Esempio con i dizionari

Errore o flusso di programma?

```
try:
        D[key] = D[key] + value
except KeyError:
        D[key] = value
```

Forse era meglio usare il metodo “giusto”:

```
D[key] = value + D.get(key, 0)
```

Leggete la documentazione!

Documentazione sulle **CLASSI**:
https://docs.python.org/3.6/tutorial/classes.html

Documentazione sulle **LISTE**:
https://docs.python.org/3/tutorial/datastructures.html

Documentazione con tutti i possibili **METODI DI OVERLOADING**:
http://www.diveintopython3.net/special-method-names.html

Documentazione con tutte le possibili **ECCEZIONI**:
https://docs.python.org/3/library/exceptions.html