Rudimenti di Python

senza rodimenti di Python

Contatti

_ Enrico Marino

_ Federico Spini

_ mail:
  _ `(marino|spini)@dia.uniroma3.it`

_ sito:
  _ `dia.uniroma3.it/~(marino|spini)/python`

Riferimenti

_ sito ufficiale: http://python.org/

_ tutorial ufficiale: http://docs.python.org/tutorial/

_ documentazione: http://docs.python.org/index.html

Paradigmi di programmazione

_ Paradigma imperativo
  _ basato sul concetto di avanzamento di stato

_ Paradigma funzionale
  _ basato sul concetto di funzione

_ Paradigma ad oggetti
  _ basato sul concetto di classe

Paradigma funzionale

_ Le funzioni sono cittadini (oggetti) di prima classe

_ Le variabili svolgono lo stesso ruolo che hanno in matematica
  _ sono la *rappresentazione simbolica di valori*
  _ servono come *parametri di funzioni*

_ Le espressioni sono l'idea sintattica fondamentale
  _ le funzioni si definiscono a partire dalle espressioni

_ La ricorsione è la struttura primaria di controllo

Paradigma funzionale
Caratteristiche principali

_ attenzione particolare sull'elaborazione delle *liste*

_ preservazione della semplicità della notazione matematica
  _ basata sulla valutazione delle espressioni

_ focus sugli *operatori* piuttosto che sugli *operandi*
  _ largo uso di *funzioni di ordine superiore*

Paradigma funzionale
Eliminare gli statement del controllo di flusso

_ Controllo di flusso basato sugli statement

```
if cond1: func1()
elif cond2: func2()
else: func3()
```

_ Espressione equivalente

```
(cond1 and func1())
    or (cond2 and func2())
    or (func3())
```

_ `func1` e `func2` non devono ritornare `False`

Paradigma funzionale
Caratteristiche funzionali

- operatore `lambda`

- `map(...)`

```
map(function, sequence[, sequence, ...])
   -> list
```

- `reduce(...)`

```
reduce(function, sequence[, initial])
   -> value
```

- `filter(...)`

```
filter(function or None, sequence)
   -> list, tuple, string
```

Paradigma funzionale
Operatore `lambda`

_ Crea funzioni anonime
  _ definibili inline nelle espressioni
  _ per aumentare la velocità di produzione del codice

```
raddoppia = lambda x: x * 2
```

invece di

```
def raddoppia(x):
    return x*2
```

Paradigma funzionale
Map

_ `map(...)`

```
map(function, sequence[, sequence, ...])
   -> list
```

_ applica la funzione a tutti gli elementi della sequenza
_ ritorna la sequenza dei risultati delle applicazioni

```
sequenza = range(1,6) # [1,2,3,4,5]
funzione = lambda x: x * 2
map(funzione, sequenza) # [2,4,6,8,10]
```

_ Per saperne di più

```
>>> help(map)
```

Paradigma funzionale
Reduce

_ `reduce(...)`

```
reduce(function, sequence[, initial])
   -> value
```

_ applica la funzione associativa di due argomenti
  _cumulativamente a tutti gli elementi della sequenza
_ riduce la sequenza ad un singolo valore

```
sequenza = range(1,6) # [1,2,3,4,5]
funzione = lambda x,y: x + y
reduce(funzione, sequenza) # 15
```

_ Per saperne di più

```
>>> help(reduce)
```

Paradigma funzionale
Filter

_ `filter(...)`

```
filter(function or None, sequence)
   -> list, tuple, string
```

_ applica la funzione di filtro a tutti gli elementi della sequenza
_ ritorna la sequenza di elementi filtrati

```
sequenza = range(1,6) # [1,2,3,4,5]
funzione = lambda x: x % 2 == 0
reduce(funzione, sequenza) # [2,4]
```

_ Per saperne di più

```
>>> help(filter)
```

Comprehension list

_ Costrutto sintattico per creare una lista basandosi su altre liste

```
S = [ x for x in range(10) ]

S = [ x for x in range(10) if x%2 == 0 ]

S = [ x*2 for x in range(10) if x%2 == 0 ]

S = [ (x, x*2) for x in range(10) if x%2 == 0 ]
```