PYTHON & PROGRAMMAZIONE FUNZIONALE

Corso di Linguaggi E Metodologie Di Programmazione

Lorenzo Ferrone

May 25, 2015

Lezione 1

INTRODUZIONE

Programma

- breve introduzione a Python;
- principi di programmazione funzionale
  - fondamenti teorici: λ-calcolo;
  - applicazione in Python
- esercizi, varie ed eventuali

Python

- home page: `www.python.org`
- download: `https://www.python.org/downloads/`
  - Se potete scegliete la versione 3.*.*
- tutorial: `https://docs.python.org/3.4/tutorial/index.html`

Programmazione funzionale

- Generico: `http://en.wikipedia.org/wiki/Functional_programming`
- Python: `https://docs.python.org/3.4/howto/functional.html`

- Python è un linguaggio **multi-paradigma**
- Supporta:
    - programmazione funzionale (ma non ***puramente*** funzionale)
    - programmazione procedurale;
    - programmazione ad oggetti;
    - ...

- Esempi di linguaggi puramente funzionali:
  - Haskell
  - Lisp
  - OCaml
- Molto meno usati in pratica

Readability counts:

- Pensato per essere facilmente leggibile (**dalle persone!**)

Batteries included:

- Nella distribuzione base c'è già praticamente tutto quello che serve:
  - `import module`

```java
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- Totale caratteri: **83** (Senza contare “hello world”)

```python
print ("hello, world!")
```

- totale caratteri: **8**

- Per indicare un blocco di codice python usa *l'indentazione*:

```
if x > 10:
    print ("maggiore")
else:
    print ("minore")
```

- Niente ";"
- Niente "{", "}"

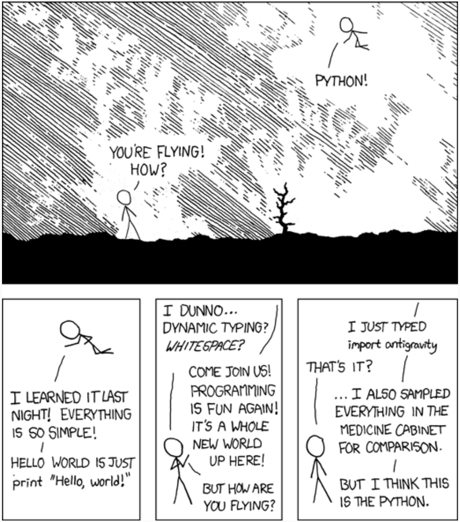

Figure 1.1: xkcd by Randal Munroe©

BASI

- IDLE editor, incluso con python
- PyCharm: (`https://www.jetbrains.com/pycharm/download/`)
- Eclipse + PyDev: `https://eclipse.org/`, `http://pydev.org/`
- Qualunque editor di testo (blocco note, Sublime Text)

- Scrivere il file:
    - ```python
      print ("hello, world!")
      ```
- salvare il file con estensione `.py`
- da terminale:

```
python script.py
```
```

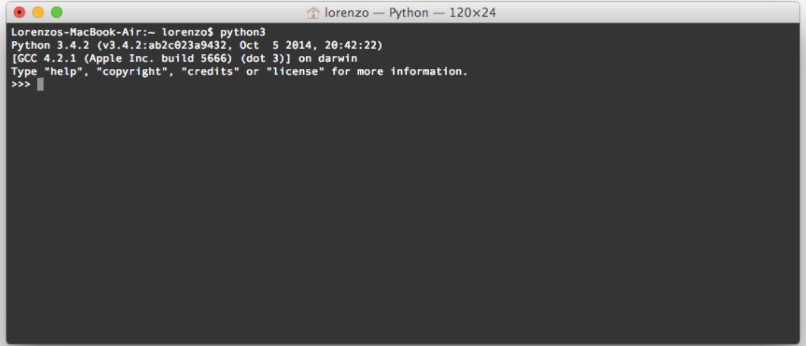

Figure 2.1: shell interattiva

TYPES

- Tipi di dati principali:
  - Numerici:
    - Interi, float, complessi
  - Booleani:
    - `True`, `False`
  - Iterabili:
    - stringhe, liste, tuple, insiemi, dizionari, ...

Non esistono le dichiarazioni dei tipi

- Interi

```
a = 1
```

- Virgola mobile:

```
pi = 3.141592653589793
```

- Numeri complessi:

```
#construction
complex_number = 7 + 4j
#get real and imaginary part
real_part = complex_number.real
imaginary_part = complex_number.imag
```

- Stringa

```
stringa = "questa è una stringa"
```

- Python supporta unicode direttamente:

```
stringa_norvegese = "åæø"
```

- Un char non è un tipo a sè stante:

```
char = "a" #è di tipo stringa (con un solo elemento)
```

- Una lista è una sequenza **ordinata**, **mutabile** di valori

```
#definition
L = [0, 1, 2, 3, 4, 5, 6, 7, 8]

#posson essere vuote
empty_list = []

#o contenere tipi diversi
mixed_list = [0, "uno", 2.0000001, [3, 4]]

#lunghezza:
len(L)
#>> 9
```

```
#definition
L = [0, 1, 2, 3, 4, 5, 6, 7, 8]

#access elements with brackets
L[0]
#>> 0

L[-1] #ultimo elemento
#>> 8

L[3:6] #dal TERZO (incluso), al SESTO (escluso)
#>> [3,4,5]
```

```
#add a single element
L.append(9)
#L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

#o un’altra lista
L.extend([10, 11])
#L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

#al contrario
L.reverse()
#L = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

#ordina, se possibile
L.sort()
```

- Altre operazioni

```
L = [0, 1, 2, 3, 4, 5, 6, 7, 8]

#un elemento è nella lista:
0 in L
# True

100 in L
# False

100 not in L
# True
```

- Altre operazioni

```
L = [0, 1, 2, 3, 4, 5, 6, 7, 8]

#rimuovi l'ultimo elemento dalla lista
elem = L.pop()
# elem = 8
# L = [0, 1, 2, 3, 4, 5, 6, 7]

#trova l'indice di un elemento nella lista
L.index(0)
# 0

L.index(100)
# ValueError: 0 is not in list
```

- Molte delle funzioni che funzionano sulle liste funzionano anche sulle stringhe

```
s = ”hello world”
len(s) # 11
s[0] # ”h”
s.index(”l”) # 2
”w” in s # True
```

- Le stringhe però sono **immutabili**

```
s = "hello world"

s[0] = "H"
# TypeError: 'str' object does not support
# item assignment
```

- Le **tuple** sono simili alle liste, ma sono **immutabili** (come le stringhe)

- coppie *chiave, valore*
- "liste" indicizzate da valori arbitrari
- equivalente di `hashmap` in Java

```
#definition
D = {"uno": 1, "due": 2, "tre": 3}

#possono essere vuoti
empty_dict = {}

#si accede agli elementi come con le liste
D["uno"] = 1

#dict non sono ordinati!
```

- Altre operazioni sui dizionari:

```
#definition
D = {"uno": 1, "due": 2, "tre": 3}

#cancellare elementi
del D["uno"]
```

OPERAZIONI

- Numeriche:

```
a = 4
b = 5

a + b  #>> 9
a - b  #>> -1
a * b  #>> 20
a / b  #>> 0.8
a % b  #>> 4
a ** b #>> 1024
```

- Su stringhe:

```
h = "hello "
w = "world"

h + w  #>> "hello world"

h.upper()
#>> "HELLO "

"HELLO ".lower()
#>> "hello "
```

- Booleane:

```
a = True
b = False

a and b  #>>False
a or b   #>>True
not a    #>>False
```

CONDIZIONALI

IF

- ```
  if x < 10:
      print ("minore")
  else:
      print ("maggiore")
  ```
- comparazioni multiple:

```
if 5 < x < 10:
    print ("compreso")
else:
    print ("fuori dall'intervallo")
```
```

ITERAZIONI

- In python il ciclo **for** permette di iterare su qualunque ***iterabile*** (liste, stringhe, tuple, dizionari, …)

```
stagioni = ["spring", "summer", "autumn", "winter"]
for stagione in stagioni:
    print (stagione)

#>> spring
#>> summer
#>> autumn
#>> winter
```

- Sui dizionari:

```
num_tradotti = {"one":1, "two":2, "three":3, "four":4}
for key, value in num_tradotti.items():
    print (key, value)

#>> "one" 1
#>> "two" 2
#>> "three" 3
#>> "four" 4
```

- Per creare cicli numerici si usa `range(n)`:

```
for i in range(5):
    print (i)

#>> 0
#>> 1
#>> 2
#>> 3
#>> 4
```

Attenzione

- In python 2 `range(n)` crea una lista
- In python 3 crea un **oggetto a sé.** (forse ne parleremo)

- `range` ha altri due parametri (valore iniziale e passo):

```
for i in range(3, 8):
    print (i)

#>> 3
#>> 4
#>> 5
#>> 6
#>> 7

for i in range(3, 8, 2):
    print (i)

#>> 3
#>> 5
```

- Esegue fino a che una condizione è vera

```
x = 0
while x < 10:
    print (x)
    x = x + 1
```

- Ci sono due modi per alterare l'esecuzione di un ciclo: **break** e **continue**

```
#ho una lista L di numeri, voglio scorrerla
#fino a che trovo un numero negativo e
#fermarmi
for num in L:
    if num < 0:
        break #esce dal loop

#oppure saltare il numero che non voglio
for num in L:
    if num < 0:
        continue
```

- Creare liste (o dizionari, tuple, etc):

```
# creare una lista di quadrati < 100
# modo classico
quadrati = []
for i in range(50):
    quadrati.append(i**2)

#meglio
quadrati = [i**2 for i in range(50)]

#possiamo aggiungere condizioni
quad_pari = [i**2 for i in range(50) if i % 2 == 0]
```

- Data una stringa, scrivere la lista di vocali di quella stringa:

```
S = "ciao mondo"
#>> ['i', 'a', 'o', 'o', 'o']

#soluzione:


vocali = [char for char in S if char in "aeiou"]
```

FUNZIONI

- Definizione:

```python
def add(a, b):
    return a + b
```

- Parametri opzionali:

```python
def add(a, b=0):
    return a + b

#posso chiamare la funzione con un solo parametro:
add(4)
#>> 4
```

- Liste arbitrarie di parametri:

```
def add(*lista_numeri):
    total = 0
    for num in lista_numeri:
        total = total + num
    return total


#chiamata
add(1,2,3,4,5,6)
#>> 21
```

- Esercizio:
  - scrivere una funzione che restituisca la norma di un vettore (in $\mathbb{R}^n$, n arbitrario)

```
import math

def norma(*coeff):
    total = sum(x**2 for x in coeff)
    return math.sqrt(total)

#chiamata
norma(1, 2)
#>> 5

#norma(1,2,3,4)
#>> 28
```

- Giusto per completezza :)

```
#definizione
class Vector2d:
#inizializzazione
    def __init__(self, x, y):
        self.x = x
        self.y = y

# nuovo vettore
v = Vector2d(3,4)

#accesso agli attributi
print (v.x)
#>> 3
```

- Per importare moduli esterni:

```
#moduli contenuti nella libreria standard:
import math, random, csv
import itertools, functools

#moduli esterni
import numpy
import matplotlib.pyplot as plt

#moduli scritti personalmente
import my_module
```

</LEZIONE 1>

1. Scrivere una funzione che produca una lista dei numeri di Fibonacci fino ad *n*, i valori di partenza possono essere arbitrari:

```
def fib(n):
    #0, 1, 1, 2, 3, 5, 8, ...
    return None
```

2. Scrivere una funzione che produca una lista di numeri primi fino ad *n*

```python
def fib(n, a=0, b=1):
    l = [a, b]
    for i in range(2, n):
        a, b = b, a + b
        l.append(a)
    return l
```

```python
def isPrime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

def primes1(n):
    primes = [p for p in range(2,n) if isPrime(p)]
    return primes
```

```python
def primes2(n):
    primes = [2]
    for i in range(3, n):
        if all(i % p for p in primes):
            primes.append(i)
    return primes
```

Lezione 2: λ-calcolo

COSA È LA PROGRAMMAZIONE FUNZIONALE

- Paradigmi di programmazione:
  - **procedurale**: lista di istruzioni che dicono al computer cosa fare
    - C, Pascal, Unix shell, ...
  - **dichiarativa**: descrizione del problema da risolvere, e il linguaggio sceglie il modo di risolverlo
    - SQL, prolog, ...
  - **a oggetti**: manipolazione di oggetti, gli oggetti hanno uno stato interno e hanno metodi per accedere e/o modificare questo stato
    - Java (obbligatorio), C++, Python lo supportano ma non è necessario.
  - **funzionale**: il problema è suddiviso in una serie di funzioni. Idealmente le funzioni dovrebbero essere mappe tra input e ouput senza altri effetti (side-effects)
    - OCaml, Haskell, (Python)

VANTAGGI

- Dimostrabilità formale
  - E' (**teoricamente!**) possibile dimostrare che un programma è corretto
- Modularità
  - Il programma viene scomposto in molte funzioni piccole che possono essere riusate facilmente
- Facilità di testing
  - Ogni funzione può essere testata indipendentemente da un'altra
- Multithreading "automatico"
  - Se le funzioni non hanno side-effects non devo preoccuparmi di lock, gestione risorse, etc

STORIA

- La programmazione funzionale si ispira al λ-*calcolo* (Alonzo Church, ~1930)
    - Sistema *formale* per studiare in maniera astratta cosa significhi *calcolare* una funzione

Fine ’800

Si inizia a sentire il bisogno di dare una formalizzazione alla matematica:

- Numeri naturali, insiemi, funzioni, …

Giuseppe Peano, 1889

Assiomatizzazione dei numeri naturali

Bertrand Russel, 1901

Paradosso di Russel: l’insieme di tutti gli insiemi che non contengono se stessi, contiene se stesso?

Ernst Zermelo, 1908

Teoria assiomatica degli insiemi

1900~1928: David Hilbert

Dare una assiomatizzazione di **tutta** la matematica:

1. **Consistente**: gli assiomi non portano a contraddizioni
2. **Completa**: tutte le proposizioni *vere* possono essere *dimostrate*
3. **Decidibile**: esiste un algoritmo per decidere quali proposizioni sono vere e quali sono false

1931: primo teorema di incompletezza di Gödel

Nessun sistema assiomatico (che contenga l'aritmetica) può essere contemporaneamente **consistente** e **completo**.

Entscheidungsproblem (*problema della decisione*)

Trovare un "algoritmo" ("effettivamente calcolabile") che permetta automaticamente di decidere quali proposizioni matematiche sono vere, e quali sono false.

Domande:

- Cos'è un algoritmo?
- Cosa vuol dire "effettivamente calcolabile"?

Rispost**E**:

- Gödel (1933): Teoria delle funzioni ricorsive generali
- Church (1934~35): λ-calcolo
- Turing (1936~37): Macchine di Turing

Risposta?

Church dimostra:

- il λ-calcolo e le funzioni ricorsive sono equivalenti
- l'*Entscheidungsproblem* **non** è risolvibile (in questi due sistemi):

> *"Non può esistere un algoritmo effettivamente calcolabile per decidere se una proposizione matematica è vera o falsa."*

Alan Turing, 1936~37

- Macchine di Turing come modello di “effettivamente calcolabile”
- l’*Entscheidungsproblem* è irrisolvibile in questo nuovo sistema
- le Macchine di Turing e il λ-calcolo sono equivalenti

Tesi di Church-Turing

Effettivamente calcolabile significa calcolabile da una macchina di Turing o nel λ-calcolo o nelle funzioni ricorsive.

λ-calcolo

- Sistema *formale* per studiare in maniera astratta cosa significhi *calcolare* una funzione:
- E' composto da:
  - λ-termini
  - regole per comporre λ-termini
  - regole di riscrittura/equivalenza tra λ-termini

λ-termini:

- variabili: $x, y, z, \ldots$
- se M è un λ-termine e x è una variabile:
  - $(\lambda x \,.\, M)$ è un λ-termine (λ- **astrazione**)
- se M, N sono λ-termini:
  - $(M\ N)$ è un λ-termine (**applicazione**)

Interpretazione:

- All'interno del λ-calcolo operiamo soltanto in maniera *formale:*
  - manipolazione automatica di simboli seconde certe regole
- Al di fuori del λ-calcolo possiamo dare un'intepretazione a cosa vogliono dire i termini e le regole

λ-astrazione:

$(\lambda x\,.\,M)$ è la definizione di una funzione che ha come input x e definizione M

Esempio:

La funzione che prende x e gli aggiunge 5:

$$\lambda x\,.\,x + 5$$

- Tradotto in codice:

```python
def f(x):
    return x + 5
```

- Unica differenza:
  - La funzione del codice ha un nome (`f`)
  - La funzione $\lambda x\,.\,x+5$ è **anonima**

Applicazione:

$(M\ N)$ è l'applicazione del termine N nell'espressione M

Esempio:

La funzione di prima applicata al numero 10:

$$(\lambda x\,.\,x + 5)\ \ (10)$$

In codice:

```python
def f(x):
    return x + 5

f(10)
```

- Due λ-termini diversi possono essere *equivalenti* (o essere trasformati l'uno nell'altro):
  - α-equivalenza;
  - β-riduzione (il "calcolo" vero e proprio di una funzione);
  - (η-conversione)

α-equivalenza:

Posso rinonimare le *variabili legate* (*bound variables*)

$$\lambda x\,.\,x \overset{\alpha}{\Longleftrightarrow} \lambda y\,.\,y$$

In codice:

```
def f(x):
    return x

def g(y)
    return y
```

β-riduzione:

Formalmente:

- Dato un termine $(\lambda x\,.\,M)\ \ (N)$:
  - sostituisco tutte le occorrenze di x all'interno di M con N;
  - levo il $\lambda x\,.$:

$$(\lambda x\,.\,x+5)\ \ (10) \overset{\beta}{\Longrightarrow} 10+5$$

Interpretazione:

- Calcolo dell'applicazione di una funzione

Currying:

- Le funzioni nel λ-calcolo prendono un solo argomento
- Per simulare funzioni a più argomenti si usa una tecnica chiamata *currying* (da Haskell Curry)

$$\lambda x y \,.\, x + y$$
$$\Downarrow$$
$$\lambda x \,.\, \lambda y \,.\, x + y$$

- In formule:

$$\begin{aligned}
\mathrm{add}\colon \mathbb{N} &\to \{f\colon \mathbb{N} \to \mathbb{N}\} \\
x &\mapsto f_x\colon \mathbb{N} \to \mathbb{N} \\
& \qquad\qquad y \mapsto y + x
\end{aligned}$$

- In codice:

```
def add(x):
    def fx(y):
        return y + x
    return fx

add(5)
#>> <function add.<locals>.fx at 0x1005ff730>

add(5)(6)
#>> 11
```

Esempio:

Come si riduce la formula:

$$(\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,x+y+z)\quad 2\;3\;4$$

?

Soluzione:

$$
\begin{aligned}
(\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,x+y+z)\ \ 2\ 3\ 4 \ &\xRightarrow{\beta}\ (\lambda y\,.\,\lambda z\,.\,2+y+z)\ \ 3\ 4\\
&\xRightarrow{\beta}\ (\lambda z\,.\,2+3+z)\ \ \ 4\\
&\xRightarrow{\beta}\ (2+3+4)
\end{aligned}
$$

CHURCH'S ENCODING

- Finora abbiamo barato:
  - abbiamo usato simboli (numeri, "+") che non appartengono al λ-calcolo
- Il vero λ-calcolo si basa **solo** sulle funzioni
- Tutte le strutture e i dati si possono *"codificare"* dentro il λ-calcolo:
  - Interi, booleani, coppie, liste, "*if-then-else*", ricorsione, …

Domanda:

Come codificare un numero n usando solo funzioni?

Risposta:

Il *numero n* viene identificato con il fatto di *“applicare n volte una funzione (qualunque) ad un valore (qualunque)”*

$$n \equiv \lambda f\,.\,\lambda x\,.\,\underbrace{f\,(f\,(f\,(f\,(\ldots\,(f}_{n\ volte}\ x)\ldots)$$

Esempi:

$$\begin{aligned} 0 &\equiv \lambda f\,.\,\lambda x\,.\,x \\ 1 &\equiv \lambda f\,.\,\lambda x\,.\,(f\,x) \\ 2 &\equiv \lambda f\,.\,\lambda x\,.\,(f\,(f\,x)) \end{aligned}$$

Successore:

$$\text{succ} \equiv \lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\,(n\,(f\,x))$$

Vediamo perché funziona calcolando esplicitamente (β-riducendo) succ **2**:

$$\begin{aligned}
\text{succ}\ \mathbf{2} &\rightarrow (\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\,(n\,(f\,x)))\quad(\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\\
&\rightarrow \lambda f\,.\,\lambda x\,.\,f\,((\lambda f\,.\,\lambda x\,.\,f\,(f\,x))\ \ (f\,x))\\
&\rightarrow \lambda f\,.\,\lambda x\,.\,f\,((\lambda x\,.\,f\,(f\,x))\ \ x)\\
&\rightarrow \lambda f\,.\,\lambda x\,.\,f\,((f\,(f\,x)))\\
&\rightarrow \mathbf{3}
\end{aligned}$$

Somma:

$$\text{add} \equiv \lambda\, m\ .\ \lambda\, n\ .\ m\ \text{succ}\ n$$

Prodotto:

$$\text{prod} \equiv \lambda\, m\ .\ \lambda\, n\ .\ m\ (\text{add}\ n)\ \mathbf{0}$$

Valori booleani

$$\text{TRUE} \equiv \lambda x \,.\, \lambda y \,.\, x$$

$$\text{FALSE} \equiv \lambda x \,.\, \lambda y \,.\, y$$

Operazioni

$$\begin{aligned} \textbf{and} &\equiv \lambda p \,.\, \lambda q \,.\, p\ q\ p \\ \textbf{or} &\equiv \lambda p \,.\, \lambda q \,.\, p\ p\ q \\ \textbf{not} &\equiv \lambda p \,.\, \lambda a \,.\, \lambda b \,.\, p\ b\ a \end{aligned}$$

</LEZIONE 2>

- Scrivere una funzione che traduca un numero n nella sua versione del λ-calcolo:

```python
def encode(n):

    return ...
```

- Implementare le funzioni di successore, somma e prodotto tra numeri in questa rappresentazione.

Lezione 3: programmazione funzionale

CONCETTI CHIAVE

- Funzioni:
  - Funzioni come oggetti base;
  - Funzioni di ordine superiore (funzioni di funzioni);
  - Funzioni ***pure***;
  - Currying
- Ricorsione;
- Lazy evaluation;

- Funzioni,
- Iteratori,
  - Generatori, **`yield`**
- `map`, `reduce`, `filter`
- **`lambda`**
- **`import operator`**
- Decoratori

FUNZIONI

- Definizione di funzione:

```
def f1(a1, a2):
    #codice della funzione
    return
```

- Le funzioni sono oggetti qualunque e possono essere definite ed usate ovunque:

```
#in una lista:
l = [f1, f2, f3]

#calcolare un elenco di funzioni in 0
for func in l:
    print (func(0))

#come parametro di una funzione
def double_application(func, x):
    return func(func(x))
```

```python
#o come valore di ritorno di una funzione
def create_func():
    def new_func(x):
        #...
    return new_func
```

MAP, FILTER, REDUCE

- Tipico della programmazione funzionale è la manipolazione di liste
- Tre funzioni tipiche:
  - `map`
  - `reduce`
  - `filter`

- Prende una lista e una funzione f e applica la funzione a tutti i valori della lista:

$$[a_1, a_2, a_3, \ldots a_n] \mapsto [f(a_1), f(a_2), \ldots, f(a_n)]$$

```python
def map(func, lista):
#...
    return
```

```python
def map(func, lista):
    return [func(x) for x in lista]
```

- Prende una lista e una funzione f e ritorna una lista con solo gli elementi per cui $f(x)$ è `True`

```
def filter(func, lista):
    l = []
    for i in lista:
        if func(i):
            l.append(i)
    return l

#equivalente a
[x for x in lista if func(x)]
```

- Prende una lista e una funzione (binaria) e applica ricorsivamente f ai valori della lista:

$$[a_1, a_2, a_3, \ldots a_n] \mapsto f(\cdots(f(f(a_1, a_2), a_3), \ldots, a_n)$$

```python
def reduce(lista, func):
    #...
```

```python
def reduce(lista, func):
    tot = lista[0]
    for elem in lista[1:]:
        tot = func(tot, elem)
    return tot
```

- Scrivere una funzione che calcoli il fattoriale di un numero:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

```python
def fattoriale(n):
    #...
    return
```

- Possibile soluzione:

```
from functools import reduce

def prodotto(a, b):
    return a*b

def fattoriale(n):
    return reduce(prodotto, range(1,n + 1))
```

- Ci sono due modi per snellire il codice:
  - `import operator`;
  - `lambda`

- Il modulo `operator` contiene la versione “funzionale” degli operatori infissi:

```
import operator
from functools import reduce

5 * 3 #>> 15
operator.mul(5, 3) #>> 15

def fattoriale(n):
    return reduce(operator.mul, range(1,n + 1))
```

- altri operatori sono:
`add, sub, mul, div, pow, not, contains, eq, lt, gt, ..`

- L'altro modo è la definizione locale di funzioni **anonime**:

```
def fattoriale(n):
    return reduce(lambda x, y: x*y, range(1,n + 1))
```

- Vi ricorda qualcosa?

```
lambda x, y: x*y
```

non è nient'altro che:

$$\lambda xy\,.\,x \cdot y$$

- Esempio: ordinare una lista in maniera non standard:

```
lista = [("Mario", "Rossi"), ("Giuseppe", "Verdi"),
("Luca","Bianchi")]

lista_ordinata = sorted(lista)
#>> [('Giuseppe', 'Verdi'), ('Luca', 'Bianchi'),
#>> ('Mario', 'Rossi')]
```

- La funzione `sorted` accetta un parametro opzionale: `key`
- `key` deve essere una funzione, la quale viene usata al posto dell'oggetto come parametro per l'ordinamento:

```
lista = [("Mario", "Rossi"), ("Giuseppe", "Verdi"),
("Luca","Bianchi")]

def get_surname(x):
    return x[1]

lista_ordinata = sorted(lista, key=get_surname)
#>> [("Luca","Bianchi"),
#>> ("Mario", "Rossi"),
#>> ("Giuseppe", "Verdi")]
```

- Usando **lambda**:

```
def get_surname(x):
    return x[1]

lista_ordinata = sorted(lista, key=get_surname)

#meglio:
lista_ordinata = sorted(lista, key=lambda x: x[1])
```

Non esagerate!

Riuscite a capire cosa fa?

```
str(reduce(lambda x,y:x+y,
    map(lambda x:x*x,range(1,1001))))[-10:]
```

E ora?

```
sum(x**2 for x in range(1, 1001)) % 10000000000
```

ITERATORI

Iteratore

Qualunque oggetto che posso iterare:

- Liste, stringhe, tuple, dizionari, …

Possono essere:

- *Lazy* o *eager* (la differenza tra `range` in python3 e python2)
    - Calcolare i valori solo quando servono
    - Rende possibile avere iteratori **infiniti**

Creare iteratori (lazy):

- Generatori
- Generator comprehension
- Combinare o modificare altri iteratori (`import itertools`)

Generatori

si definiscono con la keyword **yield**

```
def natural_numbers():
    i = 0
    while True:
        yield i
        i = i + 1

for i in natural_numbers():
    print (i)

#>> 1, 2, 3, 4, 5, ...
```

Esercizio:

Prendete l’esercizio sui numeri di Fibonacci della volta scorsa e trasformatelo in iteratore infinito.

```python
def fib(a=0, b=1):
    while True:
        yield a
        a, b = b, a + b
```

Generator comprehension

Praticamente identica alle list comprehension:

```
lista_quadrati = [i**2 for i in range(1,10)]
#>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

generatore_quadrati = (i**2 for i in range(10))
#>> <generator object <genexpr> at 0x1007739d8>

generatore_quadrati2 = (i**2 for i in natural_numbers())
```

- Il modulo `itertools` contiene funzioni per creare e manipolare iteratori:

```
import itertools

l = [1,2,3,4,5]
itertools.cycle(l)
#>> 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

```python
import itertools

l = [1,2,3,4,5]
itertools.combinations(l, 2)
#>> (1, 2), (1, 3), (1, 4), (1, 5),
#>> (2, 3), (2, 4), (2, 5),
#>> (3, 4), (3, 5),
#>> (4, 5)
```

```
import itertools

l_1 = [1,2,3]
l_2 = ["a", "b", "c"]
coppie = itertools.product(l_1, l_2)
#>> (1, 'a') (1, 'b') (1, 'c')
#>> (2, 'a') (2, 'b') (2, 'c')
#>> (3, 'a') (3, 'b') (3, 'c')
```

- Come filtro gli iteratori?
- Purtroppo sugli operatori non si può usare la sintassi delle slice...

```
import itertools

l_1 = [1,2,3]
l_2 = ["a", "b", "c"]
coppie = itertools.product(l_1, l_2)
#>> (1, 'a') (1, 'b') (1, 'c')
#>> (2, 'a') (2, 'b') (2, 'c')
#>> (3, 'a') (3, 'b') (3, 'c')

coppie[3]
#>> TypeError: 'itertools.combinations' object is
#>> not subscriptable
```

- Esiste una funzione apposita, `islice`:

```
import itertools

for i in itertools.islice(natural_numbers(), 5):
    print (i)

# >> 0
# >> 1
# >> 2
# >> 3
# >> 4
```

- Per filtrare un iteratore basandosi su una condizione ci sono:
  - `takewhile`
  - `dropwhile`
- Ritornano un iteratore che da valori fino a che una condizione è vera (o da quando una condizione è vera in poi):

```
import itertools

for i in itertools.takewhile(lambda x: x < 100, fib()):
    print (i)

# >> 0
# >> ...
# >> 89
```

Attenzione

Alcune funzioni si possono usare anche sui generatori, altre no:

```
import itertools

fib_smaller = itertools.takewhile(lambda x: x < 100, fib
sum(x for x in fib_smaller)
# >> 232

len(x for x in fib_smaller)
# >> object of type 'itertools.takewhile'
# >> has no len()
```

- Se è proprio necessario potete trasformare in lista:

```
import itertools

L = list(fib_smaller)
# >> [0, 1, 1, 2, 3, ..., 89]

len(L)
#>> 12
```

DECORATORI

Decoratori

Tecnica per modificare una funzione aggiungendone funzionalità.

Esempio:

Misurare il tempo di esecuzione di una funzione

```
import time

def func():
    #....
    return value

start = time.time()
func()
end = time.time()

print ("ci ha messo ", end - start, " secondi")
```

- Se abbiamo tante funzioni da cronometrare separatamente diventa complesso, vorremmo un metodo più generico:

```
import time

def func():
    #...
    return value

#modifico la funzione
func = time_function(func)

func()
#>> "ci ha messo 0.123 secondi"
```

```
import time

def time_function(function):
    def new_function():
        start = time.time()
        value = function()
        end = time.time()
        print ("ci ha messo ", end - start, " secondi")
        return value
    return new_function

def func():
    #...
    return value
```

- Posso usare la sintassi @decorator:

```
import time

def time_function(function):
    #....

@time_function
def func():
    #...
    return value

func()
#>> "ci ha messo 0.123 secondi"
```

</Lezione 3>

- Scrivere una funzione che calcoli la composizione di una lista arbitraria di funzioni (ad un parametro):

$$[f_1, f_2, f_3, \ldots f_n] \mapsto F$$

con:

$$F(x) = f_1(f_2(f_3(\ldots(f_n))))) \, (x)$$

```
from functools import reduce

def compose(*functions):
    #...
    return #...
```

- Scrivere una funzione `accumulate` che riduca una lista come `reduce`, ma che restituisca una lista con tutti i risultati intermedi:

```
def accumulate(function, list):
    #...
    return #...

l = [1, 2, 3, 4, 5, 6, 7]
acc = accumulate(operator.add, l)
#>> [1, 3, 6, 10, 15, 21, 28]
```

```
from functools import reduce

#definizione
def compose(*args):
    return reduce(lambda f,g: lambda x: f(g(x)), args)

#>> esempio
f = lambda x: 3*x
g = lambda y: 4 + y

C = compose(f, g, g, f)
print (C(3))

#>> 51
```

SOLUZIONI 2

```
def accumulate(func, lista):
    tot = lista.pop(0)
    l = [tot]
        for i in lista:
        tot = func(tot, i)
        l.append(tot)
    return l
```

In conclusione...

```python
import this
```

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Fine