Introduzione alla programmazione in Python: Parte I

corso tenuto da:

Francesco Grigoli

organizzato da:

Associazione Next
Studio Mirabilia

con la collaborazione di:

ANFE, Sportello multifunzionale di Bagheria

Sommario

Cosa è Python?

Python è un linguaggio di programmazione multiparadigma di altissimo livello. A differenza di linguaggi più tradizionali (ad.es C/C++, Fortran) è un linguaggio interpretato. Python è largamente utilizzato in differenti campi di applicazione:

Calcolo scientifico

Sviluppo software web (ad es. You Tube)

Sviluppo applicazioni per Smartphone

Gestione dei sistemi Unix

Sviluppo interfacce grafiche (GUI)

Perchè Python?

Negli ultimi anni la popolarità di Python è cresciuta enormemente. Le ragioni di questo grande successo sono dovute ai seguenti fattori:

Sintassi semplice (Facilità di apprendimento)

Grandissimo numero di librerie disponibili (Velocità di sviluppo)

Mancanza del processo di compilazione e linkage, (Non servono makefiles)

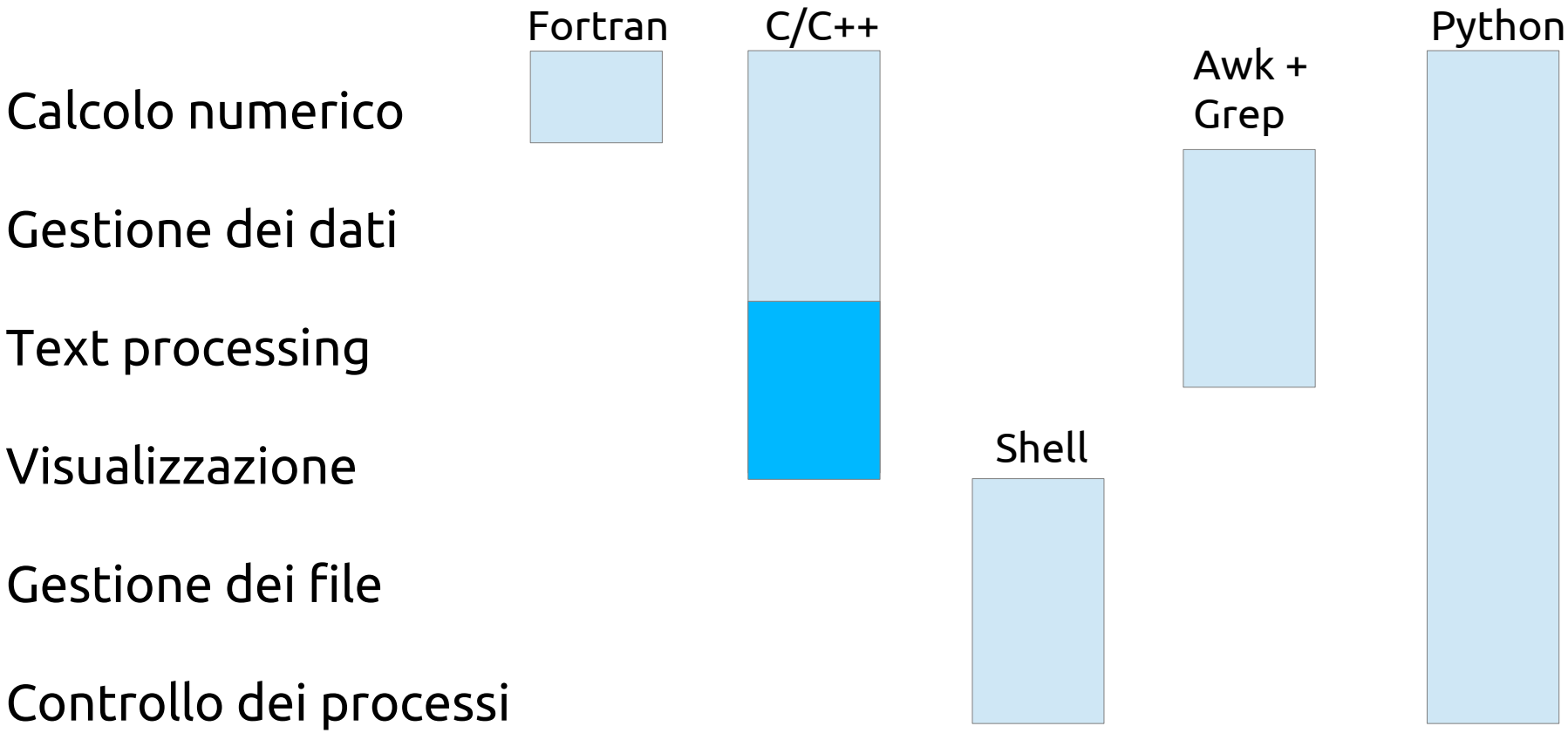
Minimalismo (Meno righe di codice = Meno errori)

Grande portabilità (indipendenza dalla piattaforma utilizzata)

Caratteristiche di Python

- “Linguaggio ad altissimo livello”
- Sintassi semplice
- Tipizzazione dinamica
- Object-oriented, multiparadigma
- Garbage collection

Differenti applicazioni - Differenti strumenti



Esempio di codice

```
import urllib, os
# Indirizzo Web
url =“http://www.geo.uni-potsdam.de/institutsmitglieder.html”
# Accede alla pagina web
page = urllib.urlopen( url ).read()
# Ricerca all'interno della pagina
if page.find(“Grigoli”) != -1:
        # Manda un messaggio a tutti i pc connessi in rete
        wall = os.popen(“wall”, “w” )
        wall.write(“Presente!!!”)
```

Uso della shell interattiva di Python

Python può essere utilizzato in modo interattivo !!!

```
> python
Python 2.5.1 ...
>>> 1+1
2
>>> print "1 + 1 =", _
1 + 1 = 2
```

Python come calcolatrice

```
>>> from math import * # per importare le funzioni matematiche di base
>>> sin(pi/2.0)
1.0
```

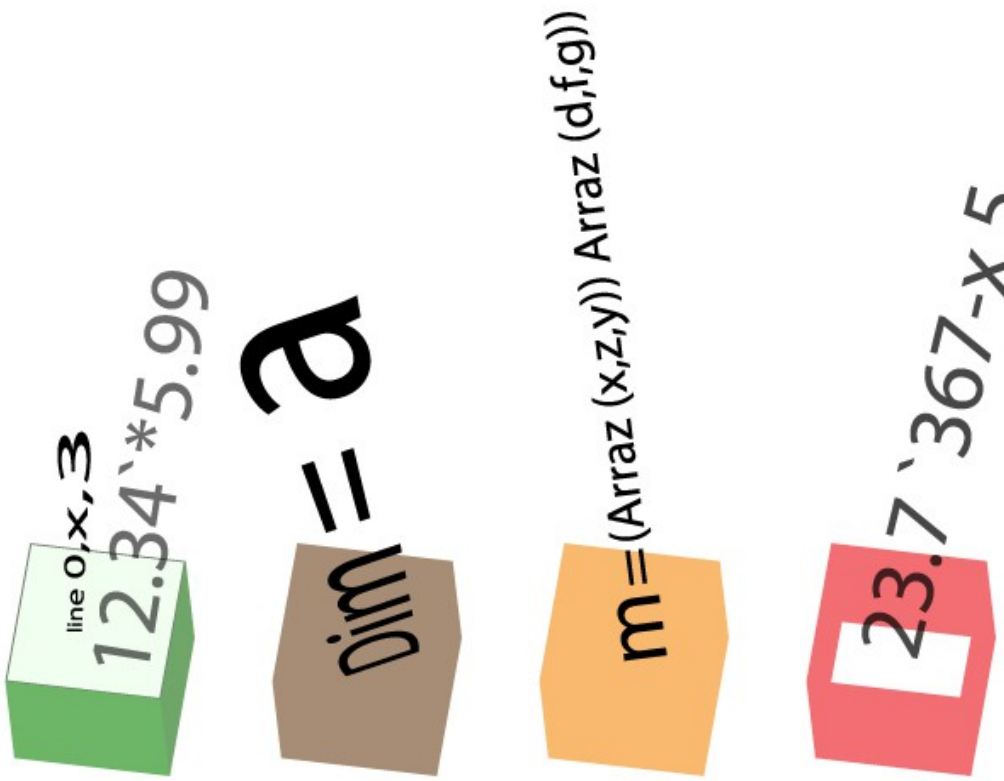
La divisione per interi restituisce floor():

```
>>> 5/2
2
```

Casting delle variabili:

```
>>> 5.0/2
2.5
>>> 5/2.0
2.5
```

Variabili



Variabili

Una variabile è creata semplicemente assegnando ad essa un valore:

```
>>> raggio_terra = 6.371e6
>>> print raggio_terra * 2.
12742000.0
```

Nota: Tecnicamente, invece di assegnare un numero ad una variabile, il termine *raggio_terra* è qui legato ad un numero immutabile. Questa è una sottile differenza con gli altri linguaggi e ritorneremo su questo argomento dopo!

Tipi di variabili

Una variabile può essere di tipo differente:

Intera (integer)

Reale (float)

Complessa (complex)

Carattere (character)

Stringa (string)

Stringhe

Le stringhe sono definite da virgolette singole o doppie :

```
>>> print ’Ciao’
Ciao
>>> print “Ciao”
Ciao
```

Numeri complessi

I numeri complessi sono definiti intrinsecamente all'interno del linguaggio:

```
>>> 1j**2
(-1+0j)
>>> a = (3+4j)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)
5.0
```

Stringhe

Le sequenze di escape sono uguali a quelle degli altri linguaggi:

```
>>> print '\'\''
''
>>> print 'X\nX'
X
X
>>> print '\\'
\
```

Stringhe: operazioni di base

Concatenazione:

```
>>> 'sp' + 'am'
'spam'
>>> 'spam' * 10
'spamspamspamspamspamspamspamspamspamspam'
```

Operazioni di base

Sottostringhe e slicing:

```
>>> cibo = 'bratwurst'
>>> cibo[0]
'b'
>>> cibo[0:1]
'b' # differente rispetto ad altri linguaggi!
>>> cibo[1:4]
'rat'
>>> cibo[-5:]
'wurst'
```

Lunghezza della stringa

Usa *len (stringa)* per restituire la lunghezza di una stringa:

```
>>> len('bratwurst')
9
```

0	1	2	3	4	5	6	7	8
B	r	a	t	w	u	r	s	t
-9	-8	-7	-6	-5	-4	-3	-2	-1

Stringhe (e numeri) sono immutabili

Le stringhe in Python non possono essere cambiate!

```
>>> cibo = 'bratwurst'
>>> cibo[0:4] = 'curry' # impossibile
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Ma, creare una nuova stringa é facile ed efficiente:

```
>>> cibo = 'bratwurst'
>>> cibo = 'curry' + cibo[-5:]
>>> print cibo
currywurst
```

Stringhe: metodi

Le stringhe sono oggetti con molti utili metodi:

```
>>> cibo = 'bratwurst'
>>> cibo.find( 'wurst' ) # cerca una sottostringa
4

>>> cibo = ' bratwurst '
>>> cibo.lstrip() # rimuove lo spazio iniziale
'bratwurst '
>>> cibo.strip() # rimove lo spazio all'inizio e alla fine
'bratwurst'
```

Ulteriori metodi

```
>>> 'Pollo Tandoori Indiano'.split()
['Pollo', 'Tandoori', 'Indiano']

>>> ' o '.join(["Carne", "Patatine", "Insalata"])
'Carne o Patatine o Insalata'
```

Ci sono molti altri metodi applicabili alle stringhe. Guardare sempre la documentazione online:

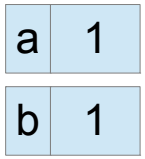
http://docs.python.org/lib/string-methods.html

Variabili vs Identificatori

In C e in Fortran, una variabile si comporta come una scatola contenente un valore:

a = 1;

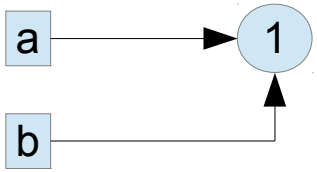
b = a;



Python, al contrario, ha identificatori (*or identifiers*), che posso essere legati a oggetti:

a = 1;

b = a;



Variabili vs Identificatori

Esempio: aggiungi due numeri

a = 1

b = 2

b = a + b

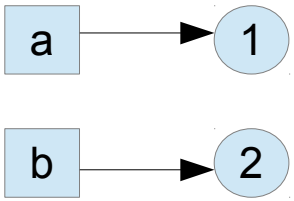
Diamo un'occhiata, più da vicino, alle operazioni...

Variabili vs Identificatori

(step 1)

a = 1

b = 2



Cominciamo con due identifiers legati da due differenti oggetti numero...

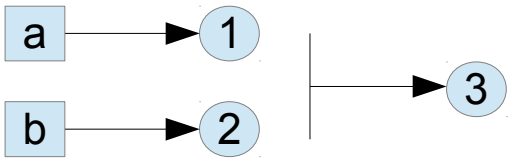
Variabili vs Identificatori

(step 2)

a = 1

b = 2

a + b



L'operatore “+” crea un nuovo numero oggetto...

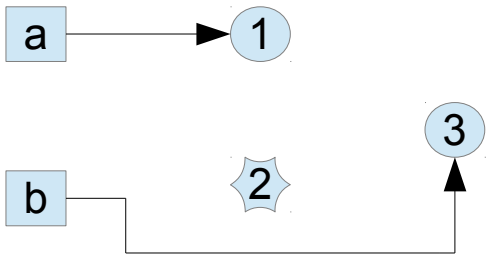
Variabili vs Identificatori

(step 3)

a = 1

b = 2

b = a + b



Alla fine, l'identifier b è associato al nuovo numero oggetto. In Python le variabili sono etichette per oggetti, tante etichette possono puntare allo stesso oggetto. (ad es. a, b e c sono etichette che possono essere associate ad uno stesso oggetto numero, per esempio 3).

Contenitori

Liste

Una lista viene creata scrivendo tra parentesi [ ]:

```
>>> Cibi = [ "Cotoletta", "Patatine", "Insalata" ]
```

Gli elementi di una lista potrebbero di diverso tipo:

```
>>> print [ 'Uno', 2, 3.0, Cibi ]
['Uno', 2, 3.0, ['Cotoletta', 'Patatine', 'Insalata']]
```

Lista vuota:

```
>>> vuota = []
```

Liste

Le liste, come le stringe, sono delle sequenze, ed hanno molte cose in comune

```
>>> Cibi = ['Cotoletta', 'Patatine', 'Insalata']
>>> len(Cibi)
3
>>> Cibi[0:2]
['Cotoletta', 'Patatine']
```

Modificare le liste

Diversamente dalle stringhe (che sono immutabili), le liste sono sequenze mutabili

```
# Rimpiazza i primi due elementi
>>> Cibi[0:2] = [ "Bratwurst", "Ketchup" ]
>>> print Cibi
['Bratwurst', 'Ketchup', 'Insalata']
# Rimuove gli ultimi due elementi
>>> Cibi[-2:] = []
>>> print Cibi
['Bratwurst']
```

Modificare le liste

```
# Inserisce nuovi elementi all'inizio
>>> Cibi[0:0] = [ "Pizza", "Pasta" ]
>>> print Cibi
['Pizza', 'Pasta', 'Bratwurst']
# ma ...
>>> Cibi[0] = [ "Pizza", "Pasta" ]
>>> print Cibi
[['Pizza', 'Pasta'], 'Pasta', 'Bratwurst']
```

Modificare le liste

Attraverso una assegnazione non si copia una lista:

(Gli Identificatori sono puntatori ad oggetti...)

```
>>> a = [1,2,3,4]
>>> b = a # a e b adesso puntano allo stesso oggetto lista
>>> b[3] = 'sorpresa!'
>>> print a
[1, 2, 3, 'sorpresa!']
>>> print b
[1, 2, 3, 'sorpresa!']
```

Modificare le liste

Con lo slicing si effettua una copia:

```
>>> a = [1,2,3,4]
>>> b = a[:] # b adesso è una copia “shallow” di a
>>> b[3] = ’sorpresa!’
>>> print a
[1, 2, 3, 4]
>>> print b
[1, 2, 3, ’sorpresa!’]
```

Usare le liste come stacks (pile)

Una lista usata come una struttura dati di tipo *last in, first out* :

```
>>> pila = [3, 4, 5]
>>> pila.append(6)
>>> pila
[3, 4, 5, 6]
>>> pila.pop()
6
>>> pila
[3, 4, 5]
```

Usare le liste come queues (code)

Una lista usata come una struttura dati *first in, first out* :

```
>>> coda = ["Eric", "John", "Michael"]
>>> coda.append("Terry") # Terry è arrivato
>>> coda.append("Graham") # Graham è arrivato
>>> coda.pop(0)
'Eric'
>>> coda.pop(0)
'John'
>>> coda
['Michael', 'Terry', 'Graham']
```

Comprensione di lista

Le comprensioni di lista sono usate per creare liste, basate sui valori delle altre liste:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Metodi delle liste da ricordare

- append(x): Aggiunge x alla fine della lista.
- extend(L): Aggiunge tutti gli elementi della lista L.
- insert(i,x): Inserisce x alla posizione i.
- pop(): Rimuove l'ultimo elemento.
- pop(i): Remuove e restituisce l'elemento alla posizione i.
- remove(x): Rimuove il primo elemento il cui valore è x.
- index(x): Restituisce l'indice del primo elemento il cui valore è x.
- sort(): ordina la lista.
- count(x): Restituisce il numero di volte che appare x.
- Reverse(): Inverte l'ordine degli elementi della lista.

Tuples

- Tuples sono liste immutabili.
- Le parentesi tonde creano tuples.
- Le parentesi in molti casi si posso omettere.

```
>>> t = 12345, 54321, 'Ciao!'
>>> t[0]
12345
>>> t
(12345, 54321, 'Ciao!')
>>> # le Tuples possono essere annidate:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'Ciao!'), (1, 2, 3, 4, 5))
```

Tuples

Si posso creare Tuples da zero o un solo elemento:

```
>>> empty = ()
>>> singleton = ('Ciao',) # <-- trailing comma! Importante! Quando la
        # tuple è  composta da un solo elemento la virgola fa capire
        # all'interprete che è una tuple e non un valore tra parentesi
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Tuples

Le tuples sono spesso usate per passare gruppi di variabili.

Packing e unpacking delle tuples:

```
>>> t = (1,2,3) # pack
>>> (a,b,c) = t # unpack
>>> print a,b,c
1 2 3
>>> a,b = a*2, b*2 # assegnazione multipla!
>>> b,a = a,b # swapping dei valori!
```

Si può spacchettare una lista in una tuple:

```
>>> (a,b,c) = [4,5,6]
>>> print a,b,c
4 5 6
```

Dizionari

- I dizionari sono usati per salvare coppie chiave-valore.
- Le parentesi graffe creano i dizionari.
- I contenuti di un dizionario non sono ordinati.
- Le chiavi (keys) sono uniche.

```
>>> colori = {'rosso': (1, 0, 0),
... 'verde': (0, 1, 0)}
>>> colori['rosso'] # visualizzo il valore
(1, 0, 0)
>>> colori['blu'] = (1, 0, 0) # aggiungo un nuovo elemento
>>> colori
{'blu': (0, 0, 1), 'verde': (0, 1, 0),
'rosso': (1, 0, 0)}
```

Dizionari

Esistono due modi per testare se un dizionario ha una key specifica:

```
>>> colori.has_key( 'rosso' )
True
>>> 'rosso' in colori
True
```

Per cancellare un elemento dal dizionario:

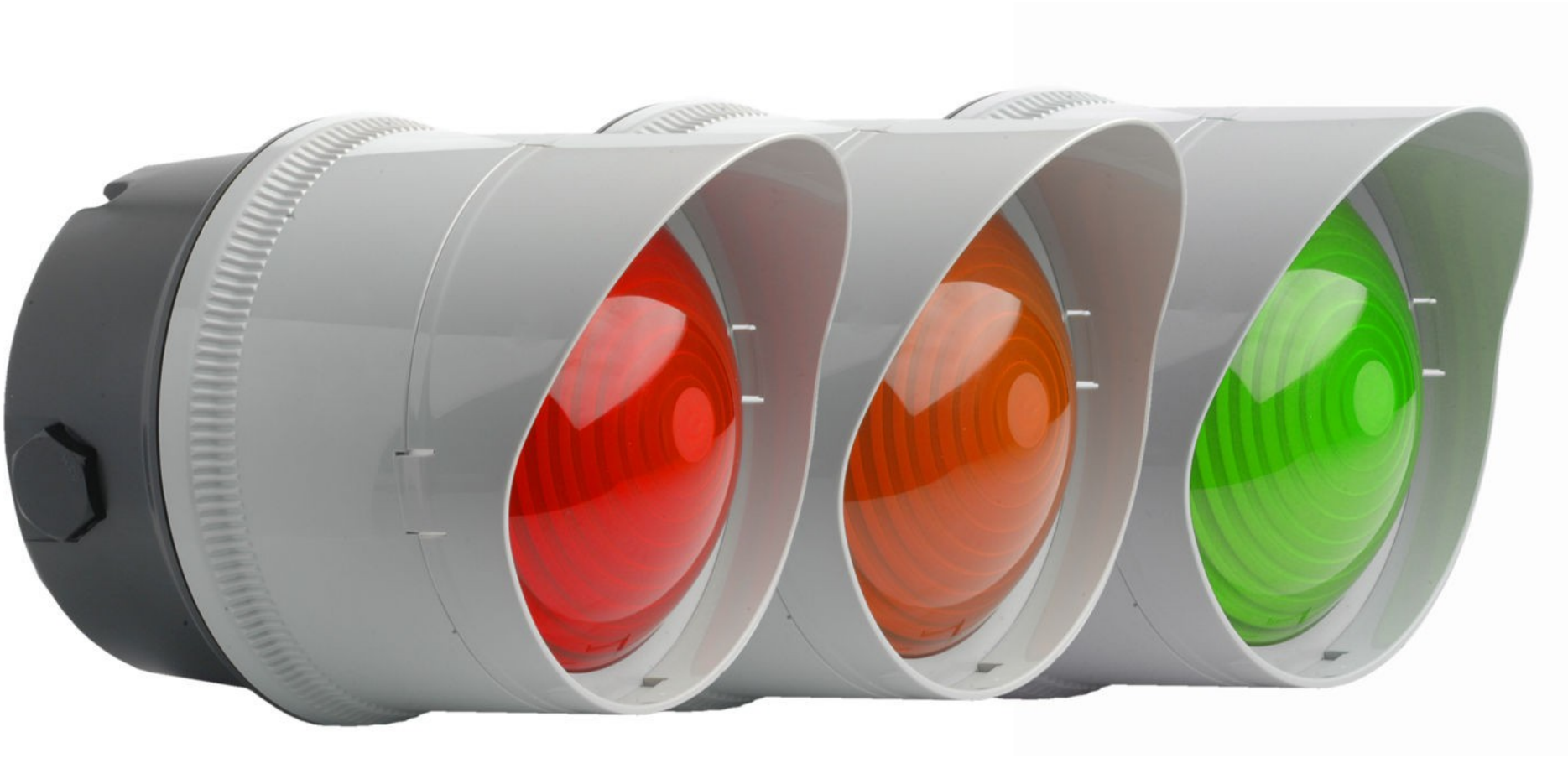
```
>>> del colori['rosso']
>>> colori
{'blu': (0, 0, 1), 'verde': (0, 1, 0)}
```

Dizionari

Per costruire un dizionario da una lista di tuples:

```
>>> dict([('Palermo', 091), ('Catania', 092), ('Messina', 090)])
{'Catania': 092, 'Messina': 090, 'Palermo': 091}
```

Controllo del flusso



Script Python Eseguibili

print_args.py :

```
#! /usr/bin/env python (non è un commento !!!)
import sys
print sys.argv
```

```
> python print_args.py 1 2 Ciao
['print_args.py', '1', '2', 'Ciao']
> chmod +x print_args.py
> ./print_args.py 2 3 Arrivederci
['./print_args.py', '2', '3', 'Arrivederci']
```

Indentazione

Python usa l'indentazione per marcare i blocchi di codice!

```
x = int(raw_input('Inserisci un intero : '))
if x < 23:
    print 'x è minore di 23'
    if x < 5:
        print 'e anche minore di 5!'
print 'the value of x is:', x
```

Indentazione

- Non mescolare tabulazioni e spazi.
- Python cerca l'esatto carattere whitespace nel tuo file sorgente
- L'editor non dice quanti spazi si stanno usando, ma è importante per Python.
- L'editor può essere impostato per utilizzare solo gli spazi, anche quando si preme TAB.

Istruzione If

```
x = int(raw_input("Please enter an integer: "))
if x < 0:
    x = 0
    print 'Negativo, impostato a zero'
elif x == 0:
    print 'Zero'
elif x == 1:
    print 'Uno'
else:
    print 'Maggiore di Uno'
```

Cosa è *true* – Cosa è *false*

- Gli interi diversi da zero sono *true*.
- Zero è *false*.
- Ogni sequenza con un numero di elementi diverso da zero è *true*.
- Sequenze vuote sono *false*.
- True è *true* mentre False è *false*.
- None è *false*.

```
import sys
file = sys.argv[1:]
if not file:  # manca l'argomento della riga di comando
        print "Errore l'argomento della riga di comando deve essere un file!."
sys.exit(1)
# Fai qualcosa con il file....
```

Il ciclo While

Vediamo fino a che punto possiamo contare su Python:

```
>>> import time
>>> i = 1
>>> while True:
... i = i * 1000 # equivalente a: i *= 1000
... print repr(i)
... time.sleep(1) # attendi un secondo
1000
1000000
1000000000
1000000000000L # <- conversione di tipo da integer a long integer!
1000000000000000L
# ... continua finchè non esaurisce la memoria disponibile!
```

Il ciclo For

Il ciclo for effettua una iterazione sugli elementi di una qualsiasi sequenza:

```
menu = ['Cotoletta', 'Patatine', 'Gelato', 'Insalata']

for cibo in menu:
    print cibo

for lettera in 'abc':
    print lettera
```

```
Cotoletta

Patatine

Gelato

Insalata

a

b

c
```

Il ciclo For

Non bisogna cambiare la lista su cui si sta operando!!!

Usa una copia della lista, in caso:

```
>>> menu = ['Cotoletta', 'Patatine']
>>> for cibo in menu[:]:
...         menu.append( cibo )
>>> menu
['Cotoletta', 'Patatine', 'Cotoletta', 'Patatine']
```

La funzione Range

Range crea una lista di numeri:

```
>>> for i in range(4):
...         print i, “-->” , 2**i
0--> 1
1 -->2
2--> 4
3--> 8
```

La funzione range si comporta in maniera simile agli operatori di slicing:

```
>>> range(4)
[0, 1, 2, 3]
>>> range(3,7)
[3, 4, 5, 6]
>>> range(2,10,2)
[2, 4, 6, 8]
```

L'istruzione Continue

L'istruzione *continue* conduce il ciclo direttamente all' iterazione successiva saltando tutto quello che viene dopo:

```
import sys
files = sys.argv[1:]
for file in files:
    if file[-4:] != '.jpg': # salta i file che non sono jpeg
        continue
# Fai qualcosa con i file jpeg
blablabla(file)
```

L'istruzione break

L'istruzione *break* interrompe immediatamente il ciclo:

```
cibo_desiderato = [ 'Cotoletta', 'Patatine' ]
menu = ['Cotoletta', 'Patatine', 'Gelato', 'Insalata']
ordinazione = []
for cibo in menu:
    if cibo in cibo_desiderato:
        ordinazione.append( cibo )
    if len(ordinazione) == 2:
        break # solo due portate, oggi...
print "vorrei ordinare : " + ' '.join(ordianzione)
```

L'istruzione pass

L'istruzione *pass* non fa niente:

```
>>> while True:
...             pass
…
```

Questo potrebbe essere usato quando è sintatticamente richiesta una istruzione ma il programma non richiede azione.

Tecniche di looping

Serve per iterare più sequenza simultaneamente:

```
>>> for a,b in zip(['1', '2'], ['3', '4']):
...         print a, b
...
1 3
2 4
```

Tecniche di looping

Serve per numerare gli elementi di una sequenza:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...         print i, v
...
0 tic
1 tac
2 toe
```

Looping di dizionari

Efficiente:

```
>>> scacchi = {“regina”: 9,  “torre”: 5, “alfiere”: 3, “cavallo”: 3, “pedone”: 1}
>>> for k, v in scacchi.iteritems():
...         print k, v
...
torre 5
regina 9
pedone 1
alfiere 3
cavallo 3
```

Looping di dizionari

```
# Sorting per chiavi:
keys = scacchi.keys()
keys.sort()
for k in keys:
      print k, scacchi[k]
# Sorting per valore:
def by_value(a,b):
     return cmp(scacchi[a],scacchi[b])

scacchi = scacchi.keys()
keys.sort(by_value)
for k in keys:
     print k, scacchi[k]
```

Ulteriori letture

- A. B. Downey, Think Python, O'Reilly
- M. Lutz, Learning Python, O'Reilly
- W. McKinney, Python for data analysis, O'Reilly
- www.numpy.org

Ringraziamenti

Desidero ringraziare Sebastian Heimann per avermi fornito le slides da cui trarre spunto e Lidia Di Blasi per averle tradotte. Ringrazio inoltre Giuseppe Gallo per aver organizzato il corso e il team dello sportello multifunzionale ANFE di Bagheria per aver messo a disposizione i loro locali.

…. Infine, grazie a tutti voi per aver partecipato...