Python

Laboratorio di Metodi Computazionale e Statistici (2023/2024)

Fabrizio Parodi

Dipartimento di Fisica

Python

- Python nasce intorno al 1990 ad opera di Guido van Rossum presso lo Stichting Mathematisch Centrum (CWI) in Olanda.
- Dopo un lungo periodo di incubazione, python inizia a diffondersi nel 2000.
- Si tratta di un linguaggio di scripting (e quindi interpretato) general purpose, in licenza Open Source.
- Python include un nutrito set di librerie, e può funzionare sia come linguaggio ad oggetti che come semplice linguaggio funzionale.
- È disponibile ed ampiamente diffuso su tutti i sistemi operativi.
- La versione corrente è python3 (python2 non è più mantenuto dal 2020)
- Manuali:
  - Python raccontato dal suo autore: `https://docs.python.org/3/tutorial/index.html`
  - Reference guide: `https://docs.python.org/3/reference/`

Python

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas.
My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers.
I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

Python

- Python è estremamente user-friendly ma è un linguaggio interpretato
- La sua collocazione miglior dovrebbe essere quella di un linguaggio di prototipaggio (come MatLab)
- La sua popolarità lo sta portando, in alcuni ambiti (Machine Learning, ad es.), ad essere il linguaggio di elezione
- Tuttavia, proprio perché interpretato la sua "impronta" ambientale è significativa. Soluzioni:
  - riportarlo ad un uso di prototipaggio
  - migliorarne l'efficienza

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

Python

Python può essere usato:

- Come shell scripting. In questo caso gli script inizieranno con

  ```
  #!/usr/bin/python
  ```

  eseguiti come un qualsiasi shell script

  ```
  ./nomepythonscript.py
  ```

  o eseguiti invocando python

  ```
  python3 nomepythonscript.py
  ```

- Oppure usato con l'apposita shell

  ```
  python3
  Python 3.8.6 (default, Feb 12 2021, 22:31:26)
  [Clang 12.0.0 (clang-1200.0.32.29)] on darwin
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
  ```

- Sessione interattiva (web-based) su `www.python.org`

Primi passi in python

```
>>> print("Hello, world")
Hello, world
```

Attenzione: python non fa differenza tra singoli ’ e i doppi apici ".
La proprietà può essere sfruttata per scrivere ’ o ". Ad esempio

```
>>> print("python all'opera")
>>> print('Il linguaggio "python"')
```

Oppure si possono usare indifferentemente ’ o " se la stringa è tra tripli apici ’’’ o """

```
>>> print("""  "python"  all'opera  """)
```

Dynamic typing, string typing

```
>>> a = 4
>>> type(a)
<type 'int'>
>>> c = 2.1
>>> type(c)
<type 'float'>
>>> s = 'abc'
>>> type(s)
<type 'str'>
>>> test = c>a
>>> print(test)
False
```

- Dichiarazione automatica delle variabili
- C'è una funzione `type` che ci dice il tipo della variabile

Operatori

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators

Liste

Python fornisce molti diversi contenitori.

Una lista è una collezione non ordinata che può essere composta da diversi tipi.

```
>>> l = [1, 2, 3, 4, 5]
>>> type(l)
<type 'list'>
>>> l[2]
3
>>> l[2:4]
[3, 4]
>>>l.append(3.1)
>>>print(l)
 [1, 2, 3, 4, 5,3.1]
```

Se dò il comando `l.` e poi dò tab (attenzione: solo in python3)

```
l.append   l.extend   l.insert   l.remove   l.sort
l.count    l.index    l.pop      l.reverse
```

Stringhe

```
>>> a = "hello"
>>> a[0]
'h'
>>> a[1]
'e'
>>> a[-1]
'o'
```

Esattamente come in C/C++

Variabili mutabili ed immutabili

- In molti linguaggi (in C/C++ ad es.) una variabile indica una locazione di memoria la cui posizione è fissa ma il contenuto variabile
- In python la filosofia è completamente diversa: "Ogni cosa è un oggetto".
- `>>> a=10`

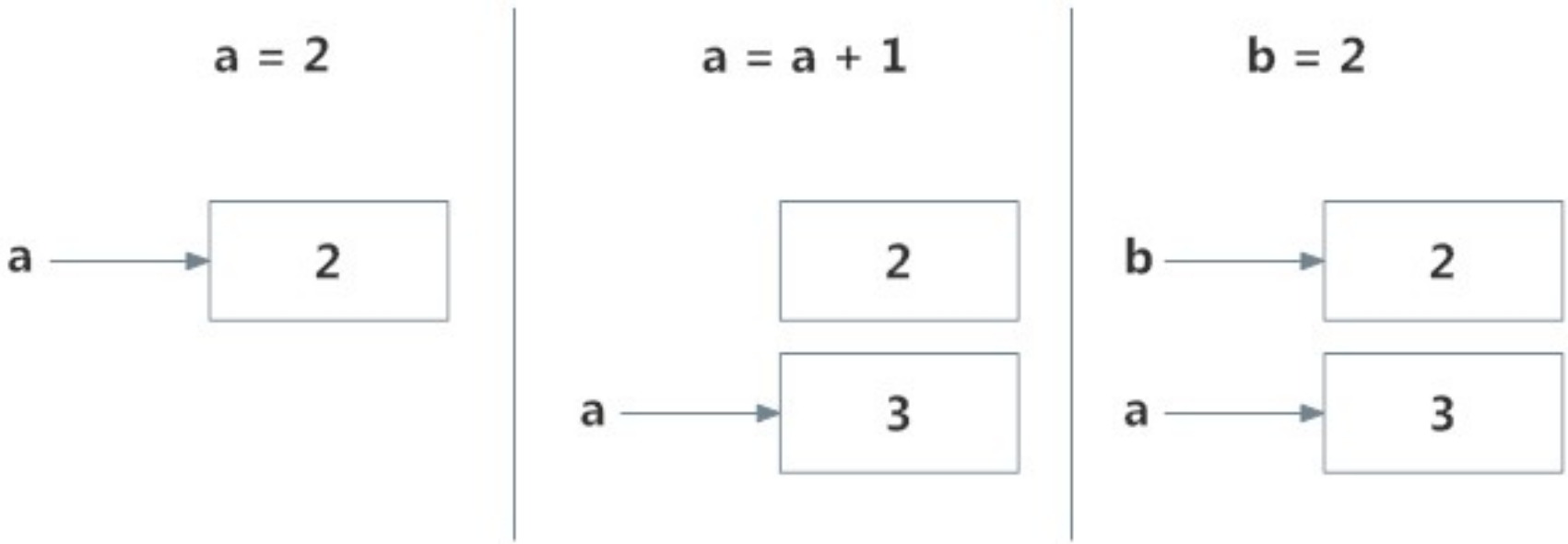
  10 è un oggetto, ma solo il nome che gli ho, momentaneamente, dato
- È quindi in python fondamentale distinguere tra oggetti
  - **immutabili** (integer, float, complex, boolean, string)
  - **mutabili** (list)

Tipi immutabili (ad es. int)

```
>>> a=2
>>> a=a+1
>>> b=2
```

Tipi immutabili (ad es. int)

```
>>> a=2
>>> a=a+1
>>> b=2
```

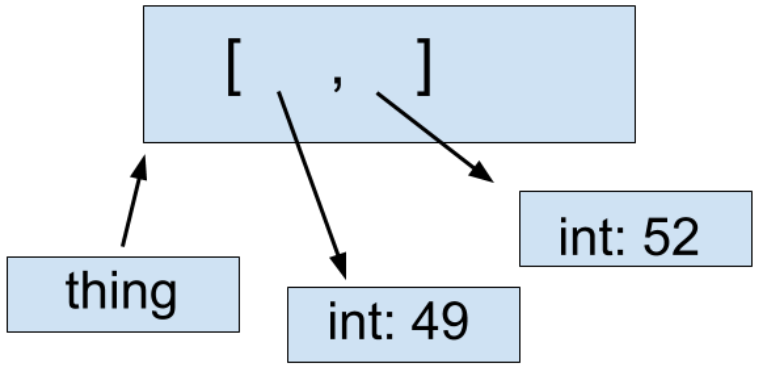


Tipi mutabili (ad es. list)

```
>>> thing=[49,52]
>>> thing.append('cat')
>>> thing[1]='dog'
```

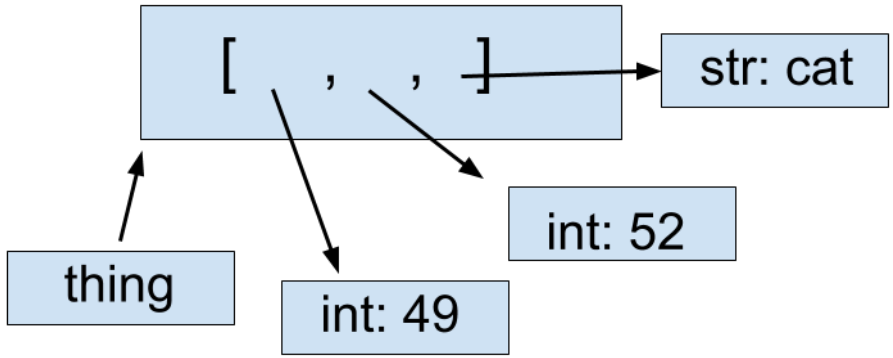
Tipi mutabili (ad es. list)

```
>>> thing=[49,52]
>>> thing.append('cat')
>>> thing[1]='dog'
```



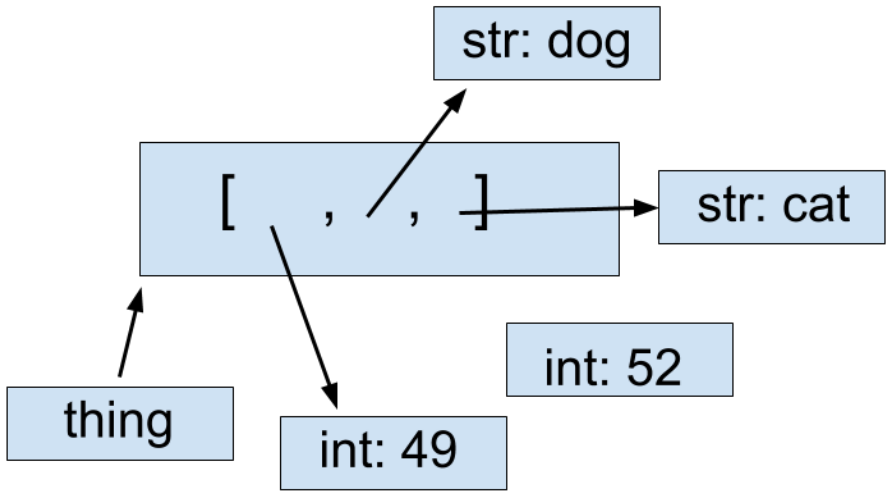
Tipi mutabili (ad es. list)

```
>>> thing=[49,52]
>>> thing.append('cat')
>>> thing[1]='dog'
```



Tipi mutabili (ad es. list)

```
>>> thing=[49,52]
>>> thing.append('cat')
>>> thing[1]='dog'
```



Gestione della memoria

- Al contrario del C++ Python gestisce autonomamente la memoria
- I seguenti oggetti sono unici (vengono creati autonomamente da Python)
  - interi tra -5 e 256
  - alcune stringhe
  - contenutori immutabili vuoti (tuples)
- Ogni altro oggetto viene creato quanto si assegna un "nome" (o una referenza) ad esso o quando viene passato ad una funzione. In Python infatti tutte le variabili sono passate per referenza all'oggetto.
- Quando una variabile non ha più "nomi" (referenze) che si riferiscono ad essa (reference count) la corrispondente porzione di memoria viene cancellata (garbage collection)

Tuple

Simile a list ma non mutabile

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = (0, 2)
```

Dictionary

Un `dictionary` è un'efficiente tabella chiave-valore (mutabile).

```
>>> tel = {'laure': 5752, 'paul': 5578}
>>> tel['francis'] = 5915
>>> tel
{'paul': 5578, 'francis': 5915, 'laure': 5752}
>>> tel['paul']
5578
>>> 'francis' in tel
True
```

Molto simile a `map` nelle STL del C++.

Mutability matters

Ma è così importante il concetto di mutabilità ? Sì e va capito per ottimizzare le prestazioni !

Ad esempio concatenazione di stringhe:
metodo 1:

```
s = ""
for data in container:
    s += data
```

metodo 2:

```
s = "".join(container)
```

Il primo approccio soffre del fatto che, siccome le stringhe sono immutabili, ogni volta viene allocato lo spazio per un nuovo oggetto il cui nome è s. Nel secondo esempio l'allocazione in memoria avviene una volta sola per la stringa finale.

if/then/else

Blocchi delimitati da indentazione

```
a = 10
if a == 1:
      print(1)
elif a == 2:
      print(2)
else:
      print('Altro numero')
```

for/range

Si può iterare con un indice

```
for i in range(0,4,1):
    print(i)
```

```
0
1
2
3
```

Il formato è `range(in,end,step)`

Stesso risultato con `range(4)`, `range(0,4)` (si possono omettere in e step).

```
for i in range(0,4,2):
    print(i)
```

```
0
2
```

for/range

Si può iterare anche su stringhe

```
for word in ('cool', 'powerful', 'readable'):
    print('Python is %s' % word)
```

```
Python is cool
Python is powerful
Python is readable
```

while/break/continue

```
z = 1+1j
while abs(z) < 100:
    if z.imag == 0:
        break
    z = z**2 + 1
```

```
a = [1, 0, 2, 4]
for element in a:
    if element == 0:
        continue
    print(1./element)
```

```
1.0
0.5
0.25
```

Funzioni

Funzione simile alla void in C++

```
def test():
        print('in test function')
test()
```

```
in test function
```

Funzione con valore di ritorno

```
def area_disco(r):
        return 3.14*r*r
area_disco(1.5)
```

```
7.064999999
```

Funzioni con parametri default

```
def square(x=2):
        return x*x
square()
square(3)
```

```
4
9
```

Funzioni parametri indirizzati per nome

```
def square(x=2,y=3,z=4):
        return x+y+z
square()
square(1)
square(1,2)
square(1,z=2)
square(x=1,y=3,4) # invalido !
```

- I parametri "posizionali" devono precedere quelli "named"
- Una volta specificati parametri con nome non ne possono più essere passati come posizionali

Funzioni con parametri multipli

```
def multiply(*args):
    z = 1
    for num in args:
        z *= num
    print(z)
multiply(1,2,3)
multiply(1,2)
```

```
6
2
```

Funzioni con più valori di ritorno

```
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt
position, velocity = yfunc(0.6, 3)
print(position, velocity)
```

Duck typing

Gli argomenti sono passati per assegnazione: in Python le variabili sono riferimenti ad oggetti; nel passaggio degli argomenti, alla variabile nella funzione viene assegnato lo stesso oggetto della variabile corrispondente nella chiamata; cioé viene fatta una copia dei riferimenti agli oggetti (quelli mutabili possono essere "mutati", gli altri no)

I tipi delle variabili vengono definiti solo all'assegnazione dei riferimenti, per cui una funzione, a priori, non sa quali sono i tipi degli argomenti ed eventuali inconsistenze producono errori solo quando la funzione viene eseguita.

In questo modo Python implementa naturalmente il polimorfismo, cioe' una stessa funzione puo' essere usata per dati di tipo diverso.

"If it walks like a duck, and quacks like a duck, it's a duck"

Duck typing

```
def sum_of(*items):
    result = items[0]
    for item in items[1:]:
        result = result + item
    print(result)
```

```
sum_of(2, 3, 5)         # -> 10
sum_of("2", "3", "5") # -> "235"
sum_of([2], [3, 5])     # -> [2, 3, 5]
sum_of(2 , b)           # TypeError
```

Variabili globali e locali

Regola generale: quando ci sono più variabili con lo stesso nome python prima cerca variabili locali, poi quelle globali poi le funzioni/variabili di sistema.

```
numbers = [2.5, 3, 4, -5]
print(sum(numbers))     # built-in Python func.
sum = 500               #  sum e' int (globale)
print(sum)
def myfunc(n):
    sum = n + 1
    print(sum)          # sum e' locale
    return sum
sum = myfunc(2) + 1     # update var. globale sum
print(sum)
```

Variabili globali e locali

Si può forzare l'uso di variabili globali con la keyword `global`

```
a = 2; b = 1      # globali

def f1(x):
    return a*x + b

def f2(x):
    global a
    a = 3
    return a*x + b

f1(2); print(a)       #stampa   2
f2(2); print(a)       #stampa   3
```

Commenti in python

I commenti in Python si possono mettere in due modi

- linea di commento

```
# questa e' una linea di commento
```

- regione commentate

```
"""
tutta
questa
regione
e' commentata
"""
```

I/O da terminale

Input

```
a = input('numero ?')
print(a)
```

Output

```
a = 10
print(a)
```

I/O da file

Input

```
f = open('workfile', 'r')
for line in f:
     print(line)
```

Output

```
f = open('workfile', 'w')
f.write('Test\n')
a=10
f.write(str(a))
```

`str` converte a in una stringa.
La funzione `repr` fornisce invece una rappresentazione univoca dell'oggetto (e anch'essa può essere ridefinita).
`str` e `repr` spesso coincidono.

Output formattato

- Old style:

```
print("a = %2.1f b= %3.1f"%(2.21,3.1))
```

- New style

```
print("a={0:2.1f} b= {1:3.1f}".format(2.21,3.1)
```

Il vecchio metodo è comunque sempre supportato. Il nuovo ha più funzionalità.

Classi

- La forma più semplice di definizione di una classe è del tipo:

```
class NomeClasse:
    <istruzione-1>
    .
    .
    .
    <istruzione-N>
```

- L'operazione di instanziazione (la "creazione" di un oggetto classe) crea un oggetto vuoto. In molti casi si preferisce che vengano creati oggetti con uno stato iniziale noto. Perciò una classe può definire un metodo speciale chiamato `__init__()`, come in questo caso:

```
    def __init__(self):
        self.data = []
```

Classi

- tutti i metodi devono avere come primo argomento l'istanza stessa (che alla chiamata non viene passata)

```
    def func(self,arg):
        self.variabile = arg
```

Esempio: classe vector

```
class vector:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __add__(self,b):
        return vector(self.x+b.x,self.y+b.y)
    def __str__(self):
        return "("+str(self.x)+","+str(self.y)+")"

a = vector(2,2)
b = vector(3,3)
c = a+b
print(c)
```

Ereditarietà

Anche in Python si può ereditare una classe derivata da una classe base.
La sintassi è:

```
class ClasseDerivata(ClasseBase):
```

Esempio: classe complex

```
class vector:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __add__(self,b):
        return vector(self.x+b.x,self.y+b.y)
    def __str__(self):
        return str(self.x)+","+str(self.y)

class complex(vector):
    def cong(self):
        return complex(self.x,-self.y)

c = complex(1,2)
d = c.cong()
print(d)
```

Moduli: come importarli

Semplice importazione di un modulo (caricamento di una certa libreria)

```
import sys
```

Importazione con cambio del nome

```
import sys  as system
```

Importazione di una sola parte del modulo

```
from functools import lru_cache
```

Importazione di tutto (senza bisogno di specificare il nome del modulo)

```
from os import *
```

Attenzione ! Se definite una funzione con lo stesso nome di una presente nel modulo "sovrascrivete" quella del modulo (perché viene usato senza utilizzarne il nome).

Moduli: sys

test.py:

```
import sys
print(sys.argv)
```

```
> ./test.py test arguments
['file.py', 'test', 'arguments']
```

Moduli: os

test.py:

```
import os
os.listdir('.')
```

```
[ 'inherit4.jpg', 'introduzione.aux', 'introduzione.log', 'introduzi
```

Moduli: numpy

numpy è il modulo per la gestione degli array N-dimensionali (simile a list ma con calcolo vettoriale (e più efficiente)). Container "mutabile".

```
import numpy as np;
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))   # Prints "<type 'numpy.ndarray'>"
print(a)         # Prints "(3,)"
a[0] = 5         # Change an element of the array
print(a)         # Prints "[5, 2, 3]"
```

Varie opzioni

- È possibile specificare il tipo di dato

```
a = np.array([1, 2, 3], dtype='f8')
```

- Si possono creare array n-dim:

```
mat = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5]] )
```

Moduli: numpy

Molte opzioni di inizializzazione:

```
import numpy as np
x1 = np.linspace(0,100,100)
x2 = np.zeros(100, float)
x3 = np.ones(100,float)
x4 = np.ones(100,float)*5
```

Moduli: numpy

Slicing:

- Significa prendere elementi da un dato indice a un altro dato indice:
  - Passiamo slice invece di indice in questo modo con [start:end]
  - Possiamo anche definire il passo, in questo modo: [start:end:step]
  - Se non passiamo start si considero 0
  - Se non passiamo end si considerato la lunghezza dell'array
- Per convenzione lo "slicing" parte da `start` ma esclude `stop` (a differenza di Matlab)

Esempio:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Moduli: numpy

Creazione array vuoto, aggiunta di un elemento alla volta (come vector)

```
import numpy as np;
a = np.array([])
for i in range(0,10):
    a = np.append(a,i)
print(a)
```