
Introduzione al linguaggio Python

Che c'entrano I serpenti?

- Nulla, il nome deriva da “Monty Python's Flying Circus” (gruppo di comici inglese)
- Show BBC ma anche film: ricordiamo tra gli altri Brian di Nazareth, Il Senso della Vita, E ora qualcosa di completamente diverso...
- Guido van Rossum (padre di Python) è un fan...

Dove trovare materiale

- Sito ufficiale del linguaggio: interprete linux / win / MacOS, IDE per Win, tutorial, reference, ...
<http://www.python.org>
- "Dive into Python" (free book molto completo)
<http://diveintopython.org/index.html>
- "How to Think Like a Computer Scientist with Python" (più introduttivo)
<http://greenteapress.com/thinkpython>
- Google -> Python

Un linguaggio interpretato

- Python, a differenza di C/C++, è interpretato (anche se poi molte implementazioni lo compilano per motivi di efficienza):
- Si può interagire con una macchina virtuale Python in maniera interattiva
- Prompt: >>>
- # questo e` un commento (come // in C++)
- Es.
- >>> 2+2
- 4
- >>> 10 / 3 # divisione intera
- 3

L'interprete

- Introducendo “#! /usr/bin/env python” come prima riga su Unix il sorgente viene eseguito senza dover manualmente invocare l'interprete
- Il simbolo “#” inizia un commento che si estende fino a fine riga
- Le istruzioni sono separate dal fine riga e non da “;”
 - Il “;” può comunque essere usato per separare istruzioni sulla stessa riga ma è sconsigliato
- Per far continuare un'istruzione anche sulla linea successiva è necessario inserire un “\” a fine riga
- Se le parentesi non sono state chiuse correttamente Python capisce che l'istruzione si estende anche sulla riga successiva

Alcuni concetti introduttivi

- Per capire il resto della presentazione serve sapere alcune cose
 - Le funzioni vengono chiamate come in C
 - `foo(5, 3, 2)`
 - “oggetto.metodo()” è un metodo (spiegati nella parte sulle classi)
 - I metodi possono inizialmente essere considerati come delle funzioni applicate sull’oggetto prima del punto
 - es.:
 - `"CiAo".lower()`

print

- L'istruzione "print" stampa il suo argomento trasformandolo in una stringa

```
>>> print 5
```

```
5
```

```
>>> print "Hello world"
```

```
Hello world
```

- A "print" possono essere passati più argomenti separati da un virgola. Questi sono stampati separati da uno spazio

```
>>> print 1, 2, "xxx"
```

```
1 2 xxx
```

- Se dopo tutti gli argomenti c'è una virgola non viene stampato il ritorno a capo

I numeri

- I numeri vengono definiti come in C:
 - “42” (intero, decimale)
 - “0x2A” (intero, esadecimale)
 - “052” (intero, ottale)
 - “0.15” (floating point, formato normale)
 - “1.7e2” (floating point, formato esponenziale)
 - Lo “0” iniziale o finale può essere omesso:
 - “.5” è “0.5”
 - “5.” è “5.0”
- Gli interi, se necessario, sono automaticamente convertiti in long (a precisione infinita)
 - `2 ** 64` → 18446744073709551616L

Gli operatori numerici

- Esistono gli stessi operatori del C, le parentesi possono essere usate per raggruppare:

```
>>> (5 + 3) * 2
```

```
16
```

```
>>> (6 & 3) / 2
```

```
1
```

- Esiste anche l'operatore elevamento “**”:

```
>>> 5 ** 2
```

```
25
```

- Non esistono “++” e “--”

Conversione fra numeri

- Esistono alcune utili funzioni di conversione fra i numeri:
 - “int(x[, base])” ritorna “x” convertito in intero. “base” è la base in cui “x” è rappresentato (il valore di default è “10”)
 - `int("13")` → 13
 - `int("13", 8)` → 11
 - `int("xxx")` → **Errore!**
 - “long(x[, base])” come “int” ma ritorna un “long”
 - “float(x)” ritorna “x” convertito in floating point
 - `float("13.2")` → 13.2
 - `float(42)` → 42.0

Le stringhe

- Le stringhe sono racchiuse fra apici singoli o doppi e si utilizzano le stesse sequenza di escape del C

```
>>> 'Python'
```

```
'Python'
```

```
>>> print "Ciao\nmondo"
```

```
'Ciao'
```

```
'mondo'
```

- Si possono usare alcuni operatori visti per i numeri

```
>>> "ciao " + "mondo"      # concatenazione
```

```
'ciao mondo'
```

```
>>> "ABC" * 3              # ripetizione
```

```
'ABCABCABC'
```

Sottostringhe

c i a o
0 1 2 3

- "ciao"[1] # carattere 1 → "i"
- "ciao"[1:3] # ⁻⁴dall'1 ⁻³al 3 ⁻²escluso ⁻¹ → "ia"
- "ciao"[2:] # dal 2 alla fine → "ao"
- "ciao"[:3] # fino al 3 escluso → "cia"
- "ciao"[-1] # l'ultimo carattere → "o"
- "ciao"[:-2] # fino al penultimo → "ci"
- "ciao"[-1:1] # non esiste → ""
- Le stringhe sono immutabili (come i numeri):
 - "ciao"[2] = "X" → **Errore!**

Altri tipi di stringhe

- Le stringhe possono estendersi su più righe, in questo caso sono delimitate da tre apici singoli o doppi

```
"""Questa è una  
stringa su più righe"""
```
- Le raw string ignorano le sequenze di escape
 - Utili per le espressioni regolari e per i percorsi su Windows
 - Non possono terminare con “\”
 - Esempi:
 - `r"C:\Windows"`
 - `r'\d.*\d$'`
 - `r'''CIAO\n'''`

Stringhe, metodi

- “S.split([sep[, max]])” ritorna una lista contenente le parti di “S” divise usando i caratteri di “sep” come separatore. “max” è il numero massimo di divisioni eseguite. Il valore di default di “sep” è ogni spazio bianco
- “S.join(seq)” ritorna una stringa contenente gli elementi di “seq” uniti usando “S” come delimitatore
- “S.lower()” ritorna “S” convertito in minuscolo
- “S.upper()” ritorna “S” convertito in maiuscolo
- “S.find(what[, start[, end]])” ritorna il primo indice di “what” in “S[start, end]”. Se la sottostirnga non è trovata ritorna “-1”
- “S.rfind(what[, start[, end]])” come “find” ma a partire dal fondo
- “S.replace(old, new[, max])” ritorna una copia di “S” con “max” occorrenze di “old” sostituite con “new”. Il valore di default di “max” è tutte.
- “S.strip()” restituisce una copia di “S” senza gli spazi iniziali e finali
- “S.lstrip()”, “S.rstrip()” come “strip” ma eliminano rispettivamente solo gli spazi iniziali e finali

Formattazione di stringhe

- L'operatore “%” serve per formattare le stringhe in modo simile alla “printf” del C
 - `stringa % valore`
 - `stringa % (valore1, valore2, valore3)`
- Le stringhe di formato sono le stesse usate dal C
 - `"-%s-" % "x"` → `-x-`
 - `"%s%d" % ("x", 12)` → `x12`
- Per mantenere il simbolo “%” si inserisce “%%”
- Si può usare la forma “%(chiave)” per inserire le chiavi di un dizionario (struttura che verrà spiegata più avanti)
 - `"%(a)d,%(b)d" % {"a": 1, "b": 2}` → `1,2`

None

- “None” è una variabile molto importante con lo stesso ruolo di “NULL” in C
- In C “NULL” è uguale a “0”
 - Rischio di confusione
- In Python “None” è di un tipo non numerico
 - Non vi è rischio di confusione con altri tipi

Liste

- Contengono elementi anche eterogenei
- Sono implementate usando array e non liste
- Per creare una lista si usano le parentesi quadre, gli elementi sono delimitati da virgole

```
>>> [1, 2, "ciao"]  
[1, 2, 'ciao']
```

- Stessi operatori delle stringhe ma sono mutabili
 - `[1] + [3, 6] → [1, 3, 6]`
 - `[1, 0] * 3 → [1, 0, 1, 0, 1, 0]`
 - `[2, 3, 7, 8][1:3] → [3, 7]`
 - `[2, 3, 7, 8][:2] → [2, 3]`
 - `[1, 2, 3][0] = 5 → [5, 2, 3]`

Liste, metodi

- Ecco i metodi più usati:
 - “L.append(obj)”, aggiunge “obj” fondo
 - “L.insert(index, obj)”, aggiunge “obj” prima di “index”
 - “L.pop([index])”, rimuove l’elemento in posizione “index” e lo ritorna, il valore di default di “index” è -1
 - “L.remove(value)”, cancella la prima occorrenza di “value”
 - “L.reverse()”, inverte la lista
 - “L.sort()”, ordina la lista
- Tutti “in place”, viene modificata la lista, non viene ritornata una nuova lista

Tuple

- Delimitate da parentesi, gli elementi sono separati da virgole
 - `(1, 2, 3)`
 - `(1,)` # un solo elemento
 - `()` # nessun elemento
- Simili alle liste ma immutabili
 - `(1, 2)[0] = 5` → **Errore!**
- Le parentesi possono essere omesse se non si creano ambiguità
 - Sono equivalenti
 - `variabile = 5, 2, 3`
 - `variabile = (5, 2, 3)`

Dizionari

- Associano ad una chiave un valore
- Creati nella forma “{chiave1: val1, chiave2: val2}”
 - {"nome": "Mario", "cognome": "Rossi"}
- L'accesso e l'inserimento di elementi avviene come per le liste
 - {"a": 1, "b": 2} ["a"] → 1
 - {"a": 1, "b": 2} ["x"] → **Errore!**
 - {} ["x"] = 2 → {'X': 2}
- Le chiavi devono essere immutabili
- Le chiavi non vengono tenute ordinate!

Dizionari, metodi

- I metodi principali dei dizionari sono:
 - “D.clear()” elimina tutti gli elementi dal dizionario
 - “D.copy()” restituisce una copia di “D”
 - “D.has_key(k)” restituisce 1 se “k” è nel dizionario, 0 altrimenti. Si può usare anche l’operatore “in”
 - “D.items()”, “D.keys()”, “D.values()” restituiscono rispettivamente:
 - Una lista con le tuple “(chiave, valore)”
 - La lista delle chiavi
 - La lista dei valori
 - “D.update(D2)” aggiunge le chiavi e valori di “D2” in “D”
 - “D.get(k, d)” restituisce “D[k]” se la chiave è presente nel dizionario, “d” altrimenti. Il valore di default di “d” è “None”

dir()

- dir() applicato a qualcosa mi dice quali nomi sono definiti in questo qualcosa (un po' vago...)
- esempio:

```
>>> a = [1, 2, 3]
```

```
>>> dir(a)['_add_', '_class_', '_contains_',  
'_delattr_', '_delitem_', '_delslice_', '_doc_',  
'_eq_', '_ge_', '_getattr_', '_getitem_',  
'_getslice_', '_gt_', '_hash_', '_iadd_',  
'_imul_', '_init_', '_iter_', '_le_', '_len_',  
'_lt_', '_mul_', '_ne_', '_new_', '_reduce_',  
'_reduce_ex_', '_repr_', '_rmul_', '_setattr_',  
'_setitem_', '_setslice_', '_str_', 'append',  
'count', 'extend', 'index', 'insert', 'pop', 'remove',  
'reverse', 'sort']
```

Variabili

- I nomi di variabili sono composti da lettere, numeri e underscore, il primo carattere non può essere un numero (come in C)
 - Sono validi:
 - “x”, “ciao”, “x13”, “x1_y”, “_”, “_ciao12”
 - Non sono validi:
 - “1x”, “x-y”, “\$a”, “àñÿô”
- Le variabili non devono essere dichiarate
- Una variabile non può essere utilizzata prima che le venga assegnato un valore
- Ogni variabile può riferirsi ad un oggetto di qualsiasi tipo

Assegnamento (1)

- L'assegnamento avviene attraverso l'operatore "="
- Non è creata una copia dell'oggetto:
 - `x = y` # si riferiscono allo stesso oggetto
- Esempio:

```
>>> x = [0, 1, 2]
```

```
>>> y = x
```

```
>>> x.append(3)
```

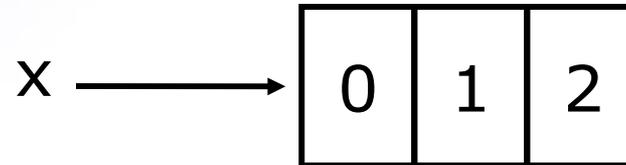
```
>>> print y
```

```
[0, 1, 2, 3]
```

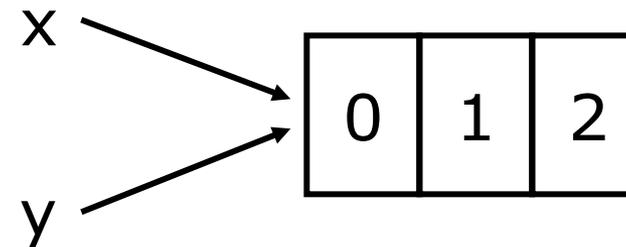
Assegnamento (2)

- Ecco quello che succede:

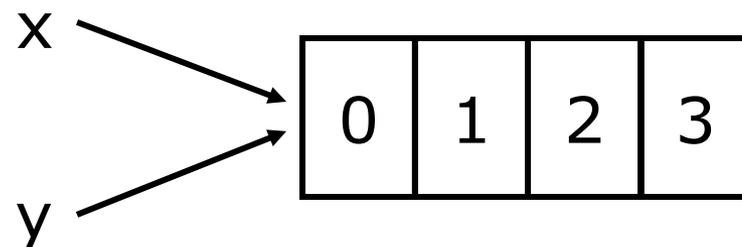
`x = [0, 1, 2]`



`y = x`

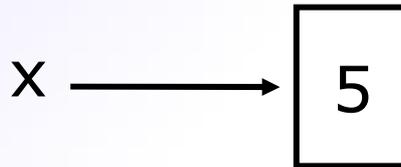


`x.append(3)`

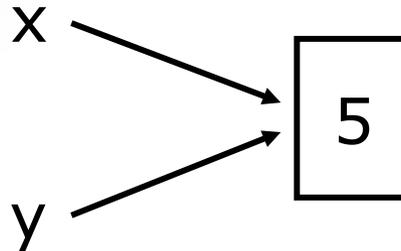


Assegnamento (3)

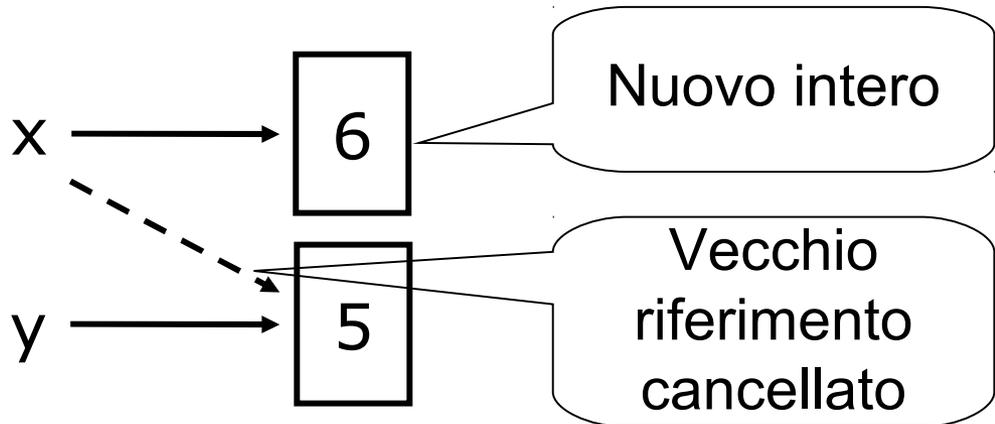
x = 5



y = x



x = x + 1



unpacking

- Un uso utile ed interessante dell'assegnamento è la possibilità di “scompattare” delle sequenze
 - $\mathbf{x, y = [1, 2]}$ $\rightarrow x = 1, y = 2$
 - $\mathbf{x, y = \{ "a": 1, "b": 6 \}}$ $\rightarrow x = \text{"a"}, y = \text{"b"}$
 - $\mathbf{x, y, z = 1, 2, 3}$ $\rightarrow x = 1, y = 2, z = 3$
- Può essere usato per scambiare i valori di due variabili
 - $\mathbf{x = 5}$
 - $\mathbf{y = 3}$
 - $\mathbf{x, y = y, x}$ $\rightarrow x = 3, y = 5$
- Il numero di elementi a sinistra deve essere uguale al numero di elementi a destra

del

- L'istruzione "del" ha due usi differenti
 - "del x" cancella la variabile "x", cioè non si potrà più usare "x" senza avergli prima assegnato un nuovo valore
 - "del" non distrugge ciò a cui la variabile si riferisce come "free" in C!
 - "del seq[ind]" cancella l'elemento con indice/chiave "ind" da "seq"

```
l = [1, 2, 3]
```

```
del l[1] → l = [1, 3]
```

```
d = {"a": 1, "b": 6}
```

```
del d["a"] → d = {"b": 6}
```

```
l2 = [1, 2, 3, 4]
```

```
del l2[1:3] → l2 = [1, 4]
```

Vero e falso

- In Python esistono due variabili “True” uguale a 1 e “False” uguale a 0
- Ogni singolo tipo o classe può definire quando il suo valore è vero o falso
- Per i tipi predefinti sono considerati falsi:
 - Il numero “0” o “0.0”
 - Una stringa vuota (“”)
 - “{}”, “[]”, “()”
- Gli altri valori sono considerati veri

Gli operatori di confronto

- Sono gli stessi del C
 - $1 == 3$ → Falso
 - $1 == 2 - 1$ → Vero
 - $1 != 2$ → Vero
 - $1 < 2$ → Vero
 - $1 > 3$ → Falso
 - $1 >= 1$ → Vero
- Se necessario vengono eseguite le necessarie conversioni intero → virgola mobile
 - $1 == 1.0$ → Vero
- Esiste anche l'operatore “<>” equivalente a “!=” ma obsoleto

Altri operatori di confronto

- “in”, vero se il primo operando è contenuto nel secondo
 - `5 in [1, 2, 3]` → Falso
 - `2 in [1, 2, 3]` → Vero
 - `"a" in {"x": 1, "a": 2}` → Vero
 - `"a" in "ciao"` → Vero
- “is”, vero se il primo operando è il secondo (non solo è uguale!)
 - Attualmente implementato come confronto fra le posizioni in memoria degli operandi
 - Usato al posto di “==” per il confronto con “None” per motivi di prestazioni

Gli operatori booleani

- “not x” 0 se “x” è vero, “1” se è falso
- “x and y” vero se sia “x” sia “y” sono veri. Ritorna:
 - Se “x” è falso lo ritorna
 - Altrimenti ritorna “y”
 - `1 and 5` → 5 → Vero
 - `[] and 1` → [] → Falso
- “x or y” vero se almeno uno degli argomenti è vero
 - Se “x” è vero lo ritorna
 - Altrimenti ritorna “y”
 - `1 or 0` → 1 → Vero
 - `() or 0` → 0 → Falso
- Sia “and” sia “or” utilizzano la logica del corto circuito
 - Il secondo argomento non viene valutato se il risultato dell’operazione è già noto in base al solo primo argomento

if

- La sintassi di “if” è:

```
if condizione:
```

```
    ...
```

```
elif condizione_alternativa:
```

```
    ...
```

```
else:
```

```
    ...
```

- Sia la parte “elif” sia la parte “else” sono facoltative
- Può esserci un numero qualsiasi di “elif”
- Non sono necessarie le parentesi intorno all’espressione booleana
- Non sono possibili assegnamenti all’interno della condizione

L'indentazione

- Il raggruppamento è definito dall'indentazione
 - Non si usano parentesi graffe, coppie “begin”/“end” e simili
 - Obbliga a scrivere codice ordinato
 - Più naturale, evita i tipici problemi del C:

```
if (0)
    printf("Questo non viene eseguito");
    printf("Questo sì");
```
- Si possono usare spazi o tabulazioni
- È *fortemente* consigliato usare 4 spazi
 - Tutti gli editor decenti possono essere configurati per sostituire il TAB con 4 spazi

pass

- Come tradurre il seguente codice in Python?

```
if (a) {}          /* Oppure "if (a);" */  
b = 12;
```

- In Python dopo un “if” deve esserci un blocco indentato

```
if a:  
    b = 12          # Dovrebbe essere indentato!
```

- Si usa quindi l’istruzione “pass” che non ha *nessun* effetto

```
if a:  
    pass           # Non fa nulla  
b = 12
```

while

- La sintassi è:

```
while condizione:
```

```
    ...
```

- Si può uscire dal ciclo usando “break”
- Si può passare all’iterazione successiva usando “continue”
- Esempio:

```
while 1: # significa ciclo infinito
    if condizione1:
        continue
    if condizione2:
        break
print 42
```

for

- La sintassi è:
`for variabile in iteratore:`
 ...
• “iteratore” può essere:
 - Una sequenza:
 - Liste
 - Tuple
 - Stringhe
 - Dizionari
 - Classi definite dall’utente
 - Un iteratore
- Si possono usare “continue” e “break” come per il “while”

Alcune funzioni built-in

- “range([start,] stop[, step])” ritorna una lista contenente gli interi in [“start”, “end”).
 - “step” è l’incremento, il valore di default è “+1”.
 - Il valore di default di “start” è “0”
 - Molto usato con i cicli “for”
 - `for i in range(5): print i`
- “len(seq)” ritorna il numero di elementi in “seq”
 - `len("ciao")` → 4
 - `len("x\0x")` → 3
 - `len([1, 2, 3])` → 3
 - `len({"a": 1, "b": 5})` → 2

Definire nuove funzioni (1)

- La sintassi è:

```
def funzione(arg1, arg2, opz1=val1, opz2=val2):  
    ...
```
- Non bisogna specificare il tipo ritornato
 - L'equivalente delle funzioni "void" del C sono funzioni che ritornano "None"
 - Una funzione può ritornare un oggetto di qualsiasi tipo
- Gli argomenti sono normali variabili e possono essere in qualsiasi numero
 - Se la funzione non accetta argomenti basta usare una lista di argomenti vuota, ad esempio:

```
def foo():  
    ...
```

Definire nuove funzioni (2)

- Gli argomenti opzionali possono non essere specificati dal chiamante, in questo caso assumono il valore di default
- Le variabili all'interno della funzione non sono visibili dall'esterno
- Esempio di utilizzo di una funzione:

```
>>> def foo(x, y, z=42, k=12):  
...     print x, y, z, k  
...  
>>> foo(5, 3, k=9)  
5 3 42 9
```

doc-string

- Le doc-string o stringhe di documentazione sono stringhe nella prima riga della funzione con lo scopo di documentarla

```
def foo() :
```

```
    "Documentazione di foo"
```

- È possibile accedere alla doc-string con l'attributo “`__doc__`” della funzione
 - `print foo.__doc__` → Documentazione di foo
- Usata da tool per generare la documentazione
- Usata dalla funzione “help”
 - “help(foo)” stampa informazioni su “foo”

return

- La sintassi è:

```
return [valore_di_ritorno]
```

- Se il valore di ritorno viene omissso viene ritornato “None”
- Se il flusso del programma esce dalla funzione senza aver trovato un’istruzione “return” viene ritornato “None”
- Esempio:

```
def somma (a, b) :  
    return a + b
```

global

- L'assegnamento all'interno della funzione assegna il valore ad una variabile locale

```
x = 5
```

```
def f():
```

```
    x = 42 # x è nuova variabile locale!
```

- Con “global nome_var” si indica all'interprete che “nome_var” è globale e non locale

```
x = 5
```

```
def f():
```

```
    global x
```

```
    x = 42 # x è la variabile globale
```

Esempio di librerie di funzioni (smtplib)

```
import sys, smtplib
fromaddr = raw_input("From: ")
toaddrs = raw_input("To: ").split(',')
msg = ''
while 1:
    line = sys.stdin.readline()
    if not line:
        break
    msg = msg + line

server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
```

OOP

- OOP = Programmazione Orientata agli Oggetti
- Incapsulamento
 - Incapsulamento di dati e codice in un unico contenitore
 - Nasconde la complessità
 - Migliora l'indipendenza
- Ereditarietà
 - Definisce le specializzazioni delle classi
 - Permette di “riciclare” il codice specializzandolo
- Polimorfismo

Classi (1)

- Una classe è simile ad una struttura del C
 - Può però contenere anche funzioni al suo interno
- La sintassi per definire una nuova classe è:

```
class NomeClasse:  
    "Documentazione della classe"  
    ...
```
- Una classe può contenere dati (variabili locali alla classe) e metodi (funzioni specifiche della classe)
- Per essere usate le classi vengono istanziate, viene cioè creata un'istanza della classe
 - Simile al concetto di allocazione di una struttura in C

Classi (2)

```
class C:(1)
    cl = 42    (2)
    def __init__(self): (3)
        self.ist = 12    (4)
```

- Viene creata una classe “C”
- “cl” è una variabile condivisa da tutte le istanze
- “__init__” è un metodo della classe “C”
 - “__init__” è un metodo speciale (chiamato costruttore), richiamato al momento della creazione dell’istanza
 - “self” è un argomento speciale che si riferisce all’istanza sulla quale è richiamato il metodo
- “ist” è una variabile locale all’istanza

Classi (3)

```
i1 = C() (1)
```

```
i2 = C() (2)
```

```
print i1.c1,i2.c1,i1.ist,i2.ist → 42 42 12 12
```

```
C.c1 = 0 (3)
```

```
i1.ist = 3 (4)
```

```
print i1.c1,i2.c1,i1.ist,i2.ist → 0 0 3 12
```

- Viene creata un'istanza di "C"
- Viene creata un'altra istanza di "C"
- Viene assegnato "0" alla variabile di classe.
 - Il cambiamento avviene per ogni istanza!
- Viene assegnato "3" alla variabile di istanza "i1.ist"
 - Il cambiamento avviene solo per "i1"!

self

- Ogni metodo accetta come primo argomento “self” che si riferisce all’istanza sulla quale il metodo è stato richiamato.

```
class C:
```

```
    def foo(self): print self, self.var
```

```
i1 = C()
```

```
i2 = C()
```

```
i1.var = 12
```

```
i2.var = 23
```

```
i1.foo()    → <__main__.C instance at 0x00931750> 12
```

```
i2.foo()    → <__main__.C instance at 0x0092BF18> 23
```

__init__

- Il costruttore viene richiamato ogni volta che viene creata un'istanza
- Il suo compito è porre la classe in una condizione iniziale utilizzabile
- Accetta "self" più eventuali altri argomenti

```
class Point:
```

```
    def __init__(self, x=0, y=0):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def stampa(self): print self.x, self.y
```

```
Point().stampa()           → 0 0
```

```
Point(1).stampa()         → 1 0
```

```
Point(y=3).stampa()       → 0 3
```

Variabili e metodi privati

- Per una buona progettazione del software è necessario che i dettagli implementativi di una classe siano nascosti
- In Python i nomi di variabili e metodo che iniziano per “__” (eccetto quelli che finiscono anche per “__”) non sono accessibili dall'esterno

```
class C: __x = 2
```

```
C().__x
```

→ **Errore!**

- Può, però, essere sufficiente lasciarli accessibili ma “segnarli” come pericolosi
- Per fare ciò basta far iniziare i nomi con un singolo “_”

Ereditarietà (1)

- Supponiamo di dover rappresentare delle figure geometriche su schermo con delle classi.
 - Tutte avranno dei metodi che fanno le stesse cose (impostano colore dello sfondo, del bordo, ecc.)
 - Solo pochi metodi differiranno (ad esempio il metodo “draw” che disegna su schermo)
- Rappresentando ogni forma con una classe separata si hanno grandi svantaggi
 - Ripetizione di codice praticamente uguale
 - Difficoltà di manutenzione
 - Ad esempio se modifico la funzione “draw” la devo modificare per tutti gli oggetti!

Ereditarietà (2)

- L'ereditarietà permette di scrivere solo il codice specifico
- Organizziamo quindi la gerarchia delle classi per l'esempio delle forme geometriche
 - “Shape”, è la classe base (o genitore)
 - Definisce le funzioni comuni (ad es. il colore dello sfondo)
 - Non definisce le funzioni specifiche di una sola classe (ad es. un metodo per ottenere il raggio)
 - Non è necessario che definisca le funzioni condivise da tutte le classi figlie, se le definisce generalmente stampa un messaggio di errore
 - “Rectangle”, è una delle classi derivate (o figlie)
 - Definisce solo i metodi che devono comportarsi diversamente da quelle del genitore (ad es. “draw”) o che il genitore non definisce
 - “Circle”, è un'altra classe derivata
 - Si comporta come “Rectangle”

Ereditarietà (3)

- La sintassi per creare una classe derivata è:

```
class Derivata(Base):
```

```
    ...
```

- Quindi realizzando l'esempio precedente:

```
class Shape:
```

```
    def set_colore_sfondo(self): ...
```

```
    def draw(self): print "Errore!"
```

```
class Rectangle(Shape):
```

```
    def draw(self): ...
```

```
class Circle(Shape):
```

```
    def draw(self): ...
```

```
    def get_raggio(self): ...
```

Ereditarietà (4)

- Ovviamente la classe può richiamare anche i metodi della classe base.
- Non può però accedervi direttamente con “self.metodo()” in quanto verrebbe chiamato il metodo della classe stessa
- Vi si può accedere con “Base.metodo(self)”
- Continuando l’esempio precedente:

```
class Triangle(Shape):  
    def set_colore_sfondo(self):  
        # operazioni specifiche di Triangle  
        ...  
        Shape.set_colore_sfondo(self)
```

- Nel costruttore è *assolutamente necessario* richiamare il costruttore della classe base!

Esempio di propagazione delle eccezioni

- Esaminiamo il seguente codice:

```
def foo(): int("X")
def bar(): foo()
bar()
```

- Ecco cosa succede:
 - Viene chiamata “bar”, “bar” chiama “foo”, “foo” chiama “int”
 - “int” incontra un errore e lancia un’eccezione
 - “foo” non gestisce l’eccezione che si propaga al chiamante
 - “bar” non gestisce l’eccezione che si propaga al chiamante
 - Il codice esterno alle funzioni non gestisce l’eccezione che si propaga all’interprete
 - L’interprete termina il programma stampando un messaggio di errore

Il messaggio di errore

- Il messaggio di errore stampato contiene diverse utili informazioni.
- Messaggio prodotto dall'esempio precedente (supponiamo si trovi nel file "f.py")

```
Traceback (most recent call last):
```

```
File "f.py", line 3, in ?
```

```
    bar()
```

```
File "f.py", line 2, in bar
```

```
    def bar(): foo()
```

```
File "f.py", line 1, in foo
```

```
    def foo(): int("X")
```

```
ValueError: invalid literal for int(): X
```

Catturare le eccezioni (1)

- Vediamo un caso tipico di gestione per gli errori:

```
try: n = int("X") # errore
except ValueError: n = 0 # valore di default
```

- Il codice all'interno del blocco "try" viene eseguito
- Se non vi sono errori il blocco "except" è saltato
- Se vi è un errore il resto del blocco "try" è saltato
 - Se il tipo dell'errore coincide con il tipo dopo la keyword "except" viene eseguito il blocco successivo
 - Se il tipo non coincide l'errore viene propagato
- Vi possono essere più blocchi "except"
 - Viene eseguito il primo il cui tipo coincide

Catturare le eccezioni (2)

- Un singolo blocco “except” può catturare eccezioni di più tipi

```
try: ...
```

```
except (NameError, TypeError, ValueError):...
```

- È possibile memorizzare l'eccezione in una variabile

```
try: ...
```

```
except (NameError, ValueError), e: print e
```

- Un blocco except senza tipo cattura tutte le eccezioni

```
try: ...
```

```
except: pass
```

- Potrebbe catturare eccezioni che non si volevano bloccare

Accesso ai file (1)

- I file vengono gestiti in modo molto semplice e simile al C
- “open(nomefile[, modo])” apre “nomefile” in modalità “modo” (“r” è il valore di default) e ritorna un oggetto di tipo “file”
- I modi sono gli stessi del C
- I metodi principali degli oggetti file sono:
 - “read([n])” ritorna “n” byte dal file. Se “n” è omesso legge tutto il file
 - “readline()” ritorna una riga
 - “readlines()” ritorna una lista con le righe rimanenti nel file
 - “write(str)” scrive “data” sul file

Accesso ai file (2)

- “writelines(list)” scrive tutti gli elementi di “list” su file
- “close()” chiude il file (richiamato automaticamente dall’interprete)
- “flush()” scrive su disco i dati presenti in eventuali buffer
- “seek(offset[, posiz])” muove di “offset” byte da “posiz”. I valori di posiz sono:
 - 0: dall’inizio del file (valore di default)
 - 1: dalla posizione corrente
 - 2: dalla fine del file (“offset” è normalmente negativo)
- “tell()” ritorna la posizione corrente
- “truncate([n])” tronca il file a non più di “n” byte. Il valore di default è la posizione corrente