

Introduzione ai Linguaggi Dinamici

Cenni storici

Gli sviluppi nei linguaggi di programmazione rispecchiano l'epoca storica e le esigenze dei programmatori.

Periodo: sistemi mainframe

- Applicazioni: orientate al calcolo scientifico
- Interfacce: predominantemente testuali
- Amministrazione: scripting per automatizzare compiti

Risultati di questo contesto:

1. **C**: concepito per prestazioni elevate, ottimo per il calcolo scientifico, ma meno portabile.
2. **Assembly**: estremamente veloce, ma scarsa portabilità; utilizzato per l'interfacciamento diretto con l'hardware.
3. **Shell**: più lento, ma altamente portabile; ideale per attività di manutenzione e scripting.

Durante questa epoca, i programmatori iniziano a riconoscere due importanti tendenze nell'architettura dei calcolatori:

1. **L'avanzamento dell'hardware**: l'hardware delle macchine si evolve in modo significativo, seguendo un ritmo simile alla legge di Moore del 1965. Questo progresso inarrestabile fornisce una potenza di calcolo crescente.
2. **Complessità in aumento**: la crescente complessità delle architetture rende sempre più difficile scrivere codice a basso livello. La gestione dei dettagli intricati richiede una quantità significativa di tempo e sforzo.

In risposta a queste sfide, emerge il concetto di "linguaggio ad alto livello general purpose", con es. significativi tra cui:

- **Python**: famoso per la sua semplicità e leggibilità del codice.
- **Ruby**: con una forte enfasi sulla programmazione orientata agli oggetti.
- **Java**: notorio per la portabilità del codice attraverso piattaforme diverse.

Nel 1991, il World Wide Web (**WWW**) venne inventato, segnando l'inizio di una rivoluzione dell'informatica.

Al fine di sviluppare **applicazioni Web**-based in modo più semplice rispetto al linguaggio C, si assiste all'**emergere di nuovi linguaggi**:

- **PHP** (PHP Hypertext Preprocessor): Nasce nel 1994 con l'obiettivo di creare pagine Web dinamiche
- **JavaScript (Mocha)**: viene introdotto nel 1995 per essere utilizzato in Netscape, uno dei primi browser Web

L'adattamento dei linguaggi esistenti: linguaggi come **C** e **Perl** vengono adattati per non perdere terreno e vengono comunemente utilizzati per la creazione di programmi CGI (Common Gateway Interface) per la gestione dinamica delle pagine Web.

Tramite <https://pypl.github.io> possiamo vedere la popolarità dei vari linguaggi. Essa cambia nel tempo ed è influenzata da mode, esigenze e politiche. Attualmente Python è il più popolare.

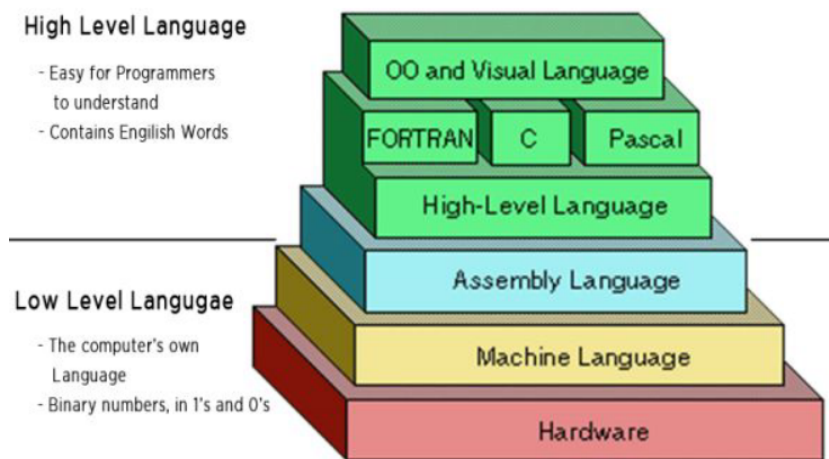
Linguaggi Statici

Un linguaggio di programmazione statico, o "linguaggio di programmazione con tipizzazione statica", è un tipo di linguaggio di programmazione in cui i tipi delle variabili e delle espressioni sono verificati a tempo di compilazione anziché a tempo di esecuzione. Tipicamente: linguaggi ad alto livello | linguaggi compilati.

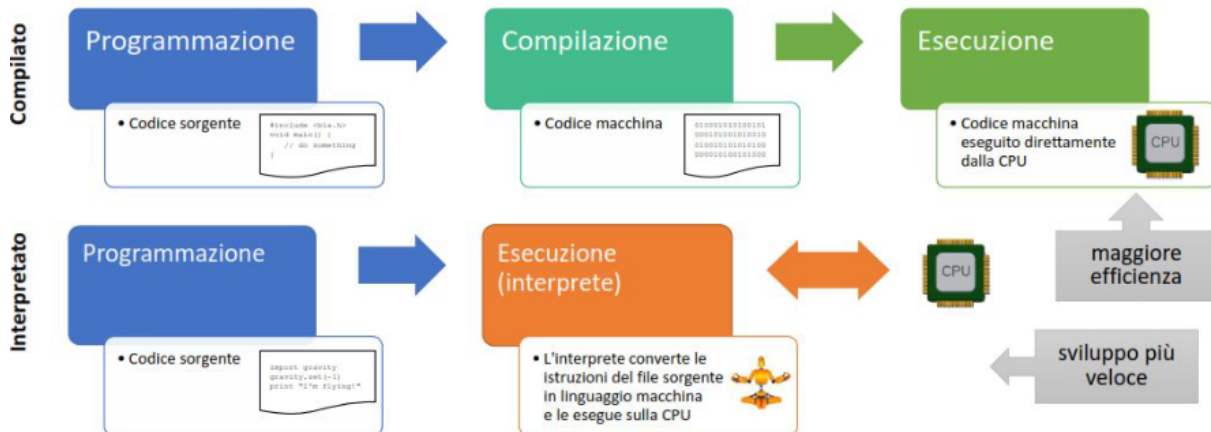
Linguaggi Dinamici

Un linguaggio di programmazione dinamico, o "linguaggio di programmazione con tipizzazione dinamica", è un tipo di linguaggio di programmazione in cui i tipi delle variabili e delle espressioni sono verificati a tempo di esecuzione anziché a tempo di compilazione. Tipicamente: linguaggi ad alto livello | linguaggi interpretati.

Linguaggi ad alto livello / a basso livello



Linguaggi Interpretati / Compilati



Pro dei Linguaggi Dinamici

- **Tipizzazione dinamica:** in un linguaggio dinamico, non è necessario dichiarare esplicitamente il tipo di una variabile durante la sua creazione. Il tipo di una variabile può cambiare durante l'esecuzione del programma.
- **Controllo dei tipi a runtime:** i controlli sui tipi e le conversioni vengono effettuati durante l'esecuzione del programma.
- **Flessibilità:** tendono ad essere più flessibili e meno rigidi rispetto ai linguaggi con tipizzazione statica.
- **Maggiore facilità di debug:** poiché i tipi vengono verificati a runtime, è possibile esaminare più facilmente il comportamento del programma durante il debug.
- **Sviluppo più rapido** e, tipicamente, **sintassi più semplice.**

Contro dei Linguaggi Dinamici

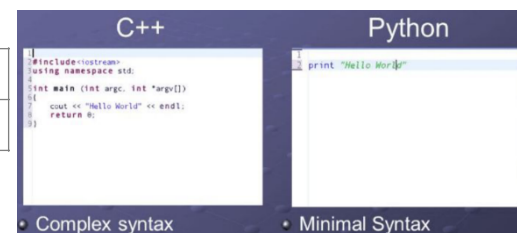
- **Prestazioni inferiori:** i linguaggi dinamici tendono ad essere più lenti dei linguaggi con tipizzazione statica. Questo perché la verifica dei tipi e l'allocazione della memoria possono comportare un overhead significativo durante l'esecuzione del programma. Le ottimizzazioni possono mitigare questo problema, ma in generale i linguaggi dinamici sono meno efficienti in termini di velocità rispetto ai linguaggi statici.
- **Errori a runtime:** poiché i controlli dei tipi avvengono a runtime, gli errori di tipo possono emergere solo durante l'esecuzione del programma. Questo rende il debug più complesso e può comportare la scoperta di errori solo quando il programma è in esecuzione, il che può essere problematico in applicazioni critiche.
- **Difficoltà nella manutenzione del codice:** a causa della tipizzazione dinamica, la manutenzione del codice può essere più complessa. È possibile che i cambiamenti al codice siano difficili da tracciare e da comprendere, soprattutto in progetti di grandi dimensioni. La mancanza di informazioni sui tipi può rendere il codice meno auto-documentante.
- **Maggiore probabilità di bug:** a causa della mancanza di verifica statica dei tipi, è più facile commettere errori legati ai tipi in linguaggi dinamici. Questi errori possono non emergere fino a quando il programma viene eseguito, portando a comportamenti imprevisti.

Caratteristiche dei Linguaggi Dinamici

- Può avere una fase di compilazione, in cui il codice sorgente viene tradotto in un formato intermedio indipendente dall'architettura (es. **bytecode** – Java). Il formato intermedio è interpretato (linguaggio portabile).
 - L'interprete si serve di funzioni interne per gestire memoria ed errori in modo automatico a runtime (assenza di dettagli ostici per il programmatore).
 - Ha una tipizzazione dinamica dei dati (possono mutare a runtime).
- Ha la caratteristica di sapersi “analizzare” e “modificare” durante l'esecuzione (**metaprogramming**)
 - Eseguire funzioni diverse a seconda delle condizioni operative a runtime
 - Cambiare il codice stesso del programma
 - Creare strutture dati variabili nel tempo
- Presenza massiccia di librerie esterne facilmente utilizzabili per diversi compiti
(es. servizi di calcolo scientifico, interfacce grafiche complesse, supporto per il Web, ...)

Python – linguaggio dinamico più utilizzato

| | | | |
|--|-------------------------------|--------------|------------|
| Linguaggio dinamico di riferimento per la piattaforma di servizi offerti da Google | | | |
| Popolarità in ascesa | Curva di apprendimento ripida | Flessibilità | Semplicità |



Tipizzazione

Un programma manipola dati attraverso istruzioni

- ⇒ **Tipo**: indica cosa rappresenta il dato (e di conseguenza che operazioni sono permesse)
- ⇒ **Istruzioni**: indicano l'operazione da compiere su dati di un certo tipo.

La stessa istruzione (da un punto di vista sintattico) può assumere diverse semantiche in base al tipo di dato; quindi, esso è fondamentale per determinare il risultato di un'istruzione (es. $1+3$ e $'a'+b'$ sono diversi)

All'interno del programma, un dato viene identificato con un **nome**.

Il programmatore utilizza il nome ma deve sapere:

- Il tipo (per non indicare operazioni sbagliate)
- Dimensione in memoria (per non saturarla)
- Scope e tempo di vita dell'entità (per sapere quando utilizzarla)

Obiettivi della tipizzazione

- ⇒ **Type Check** → l'uso della tipizzazione permette di scoprire codice privo di senso/illecito

Nei linguaggi compilati viene fatto a compile time → si prevengono errori dovuti ai tipi

Es. in C → $3/"ciao"$ non è consentito (divisione tra intero e stringa) – il programma non compila

Alcune espressioni vengono automaticamente convertite (casting) – es. $'a'/5$

- ⇒ **Ottimizzazioni**

- Alcune operazioni possono essere eseguite in maniera più efficiente (es. x^2 con uno shift di bit)
- Per eseguire tali ottimizzazioni è necessario che il compilatore/interprete sappiano il tipo di dato

- ⇒ **Astrazione** → la tipizzazione è utile anche al programmatore, che non deve preoccuparsi di come sia rappresentato il dato (se non in casi estremi)

- Non deve preoccuparsi che una stringa in realtà siano byte
- Non deve preoccuparsi se la macchina usi la rappresentazione little/big endian

- ⇒ **Interfacce** → la tipizzazione è utile per definire interfacce

- Gli argomenti tipizzati sono più esplicativi
- Prevengono il passaggio di dati sbagliati

Type Check

Type Check nei linguaggi dinamici (ricordando che uno dei loro pro è avere **tipizzazione dinamica**)

- Avviene a runtime (non è possibile determinare a priori il tipo di dato)
- ✓ **Più flessibile** (minor gestione di tutti i casi possibili)
- ✗ Maggior overhead
- ✗ Possibilità di errori a runtime
- ✗ Possibilità di comportamenti non previsti (necessità di maggior controlli e di gestione degli errori)

Type Check (recap)

- ⇒ **Statico**: identifica errori a tempo di compilazione, previene errori a runtime, più performante;
- ⇒ **Dinamico**: più flessibile (costrutti illegali in linguaggi statici, es. $y = 5$; $y = 'ciao'$), più rapida prototipazione;

Tipizzazione

Tipizzazione forte

Il linguaggio di programmazione impone rigorosamente regole sulla conversione dei tipi di dati e sulla compatibilità dei tipi:

- ⇒ Conversioni esplicite
- ⇒ Operazioni solo tra tipi compatibili

Tipizzazione debole

Il linguaggio può consentire conversioni implicitamente tra tipi di dati e può essere più permissivo nella gestione dei tipi:

- ⇒ Conversioni implicite
- ⇒ Operazioni permesse tra tipi incongruenti

Es. $a = 3$; $b = '58'$; $a + b = ?$

- In C, b è un puntatore → aritmetica dei puntatori
- In Java, a è convertito in stringa → $'358'$
- In Perl, b è convertito in intero → 61
- In Python → **Errore**

Tipizzazione safe

Un linguaggio di programmazione è considerato adottare una tipizzazione safe dei dati se **impedisce** che un'operazione di casting implicito causi un crash.

- Es. $a = 3$; $b = '58'$; $a + b = ?$ → in Perl, b è convertito in intero → 61

Tipizzazione unsafe

Un linguaggio di programmazione è considerato adottare una tipizzazione unsafe dei dati se **non impedisce** che un'operazione di casting implicito causi un crash.

- Es. $a = 3$; $b = '58'$; $a + b = ?$ → In Python → **Errore**
 - In C, b è un puntatore → aritmetica dei puntatori (fuori range)

Tipizzazione in Python

Python è un linguaggio **completamente orientato agli oggetti** → ogni variabile (che ha un tipo) è un oggetto (comprese le primitive). Nei linguaggi ad oggetti, ogni oggetto dispone dei metodi e degli attributi della sua classe e quelli ereditati.

- ⇒ Tramite l'ereditarietà si possono realizzare comportamenti diversi per uno stesso metodo

Esempio (polimorfismo)

```
class Moto extends Veicolo
    getRoute(): return 2

class Auto extends Veicolo
    getRoute(): return 4

Veicolo v = getVeicolo() //ritorno o Auto o Moto
v.getRoute() // risultato?
```

È necessario capire quale metodo effettivo chiamare (da un punto di vista del compilatore/interprete):

- ➔ Controllare che la classe abbia tale metodo (e che sia una classe/sottoclasse di Veicolo)
- ➔ Se non è così, passare alle superclassi alla ricerca del metodo
- A tempo di compilazione → più efficiente ma meno flessibile
- A tempo di esecuzione → meno efficiente ma più flessibile

Duck Typing

L'alternativa all'esempio precedente, meccanismo che è **utilizzato in Python**, è quello di non controllare il tipo, ma solo che l'oggetto abbia tale metodo (non controllo la classe di appartenenza).

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")

class Person:
    def quack(self):
        print("The person imitates a duck.")

def in_the_farm(a):
    a.quack()
```

Esempio

```
function calcola(a,b,c) => return (a+b)*c

e1 = calcola(1,2,3)
e2 = calcola([1,2,3],[4,5,6],2)
e3 = calcola('mele ', 'e arance', 3)
```

La funzione *calcola* deve ricevere dei parametri che supportino i metodi + e *

Output:

```
e1 → 9
e2 → [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
e3 → "mele e arance mele e arance mele e arance"
```

Python

Python ([docs](#)) è stato creato nel 1991 da Guido Van Rossum

- Semplicità ed estensibilità;
- Sintassi concisa e chiara;
- Esalta la leggibilità del codice;
- Archivio di moduli vasto e stabile;

Python è un:

- ⇒ **Linguaggio ad alto livello**
- ⇒ **Linguaggio dinamico**
- ⇒ **Linguaggio interpretato**
 - Modello a compilatore + interprete
 - Modello a Bytecode
 - **Portabile**
 - **Macchina Virtuale Python (PVM)**

Sistema di riferimento del corso:

- Python3
- OS: GNU/Linux, Debian based distros (Ubuntu, Mint, ...)

Comandi basilari da terminale:

- **python** → shell interattiva
- **python -c 'statement'** → lanciare un comando da linea di comando
- **python <nomefile>** → lanciare un programma python
- **./file** → lanciare direttamente un file con permessi di esecuzione
 - Il file deve iniziare con la riga **#!/usr/bin/python**
 - Indica quale interprete utilizzare per eseguire il file
- **help(argomento)** → trovare info su una funzione/modulo (documentazione integrata) – Es. *help(dict)*

Tipi base

Tipi built-in (alcuni godono di una sintassi agevolata):

- **Numerics** → valori numerici (int, float, complex)
- **Sequence** → sequenze di oggetti (str, list, tuple)
- **Set** → insiemi (set, frozenset)
- **Dict** → dizionari di coppie *key* → *value*

Numerics

- Numeri interi → lunghezza arbitraria (no limiti)
- Numeri float → precisione dipendente dall'architettura, separatore parte intera/decimale (".")
- Numeri complessi → [numero reale +] numero reale con suffisso j
 - $z = 10 + 20j$; $z = -4j \Rightarrow$ **z.real** (parte reale) e **z.imag** (parte immaginaria)
- Booleani → considerati sottotipo degli interi

Operazioni

- Operazioni aritmetiche standard: +, -, *, /
- Divisione: / - Divisione intera: //
- Resto divisione tra interi: %
- Valore assoluto: *abs()*
- Numero complesso coniugato: *conjugate()*
- Elevamento a potenza: *pow()*, **
- Arrotondamento: *math.trunc()*, *math.floor()*, *math.ceil()*, *round()*

Sequence (sequenza ordinata di elementi)

- Stringhe → racchiuse tra apici " o '", possibilità di usare i caratteri speciali (es. \n, \t, \\, ...) - sono **immutabili**
- Liste → elementi di qualsiasi tipo (anche non coerenti) racchiusi tra []
- Tuple → liste immutabili, elementi racchiusi tra ()

Operazioni (per tutti i tipi)

- Accesso → ad un elemento $a[1]$, a sottoliste/sottostringhe $a[1:3]$, partendo dal fondo $a[-1]$
- Concatenazione → $'a' + 'b' = 'ab'$, $[1,2] + [3,4] = [1,2,3,4]$
- Ripetizione → $'a'*4 = 'aaaa'$, $[1]*4 = [1,1,1,1]$
- Lunghezza della sequenza/lista → $len(a) = 4$

Stringhe – Operazioni

- *s.lower()*, *s.upper()* → ritornano una copia della stringa *s* con lettere minuscole/maiuscole
- *s.count(substr)* → ritorna il numero di occorrenze della sottostringa *substr* in *s*
- *s.find(substr)* → ritorna l'indice della prima occorrenza della sottostringa *substr* in *s*

- `s.replace(sub1,sub2)` → rimpiazza le occorrenze della sottostringa `sub1` con `sub2` in `s`
 - o Ritorna la stringa modificata, quella di partenza rimane inalterata (sono **immutabili**)
- **Join** → concatena diversi elementi aggiungendo un separatore (es. `" ".join([1,2,3]) = "1; 2; 3"`)
- **Split** → divide una stringa in elementi considerando un separatore (es. `'1 + 2 + 3'.split('+') = ['1','2','3']`)

Liste – Operazioni

- `lista.append(oggetto)` → appende l'oggetto in fondo alla lista
- `lista.insert(indice,oggetto)` → inserisce l'oggetto nella posizione indicata dall'indice
- `lista.pop(indice)` → estrae l'oggetto in posizione `indice` dalla lista
- `lista.pop()` → estrae l'ultimo elemento della lista
- `lista.sort()` → ordina gli oggetti contenuti – modifica lista in-place
- `sorted(lista)` → non modifica la lista originale ma la restituisce ordinata
- `len(lista)` → ritorna il numero di elementi contenuti in una lista
- Operatore **in** → ricerca elemento in una lista (es. `6 in lista → True`)

Coda: si usano le operazioni `lista.append(oggetto)` e `lista.pop(indice)`

Stack: si usano le operazioni `lista.append(oggetto)` e `lista.pop()`

Operazioni di rimozione

- `lista.pop(ind1)` → rimuove l'elemento di indice `ind1` e lo ritorna
- `lista.remove(elem1)` → rimuove l'elemento `elem1` (matching) senza ritornarlo
- `del lista[ind1]` → statement che rimuove l'elemento di indice `ind1` (opera anche sui range)

Operazioni di slicing

`wt = [1,2,3,4,5]`

- Base: `wt_slice = wt[1:3] → [2,3]` (notazione `[start:stop]`)
- Con incremento: `wt_slice = wt[1:5:2] → [2,4]`

Copie

Se si usa `=` si copia il **riferimento** (es. `a = [1,2,3]; b = a; b[0] = 2 → b è [2,2,3], a è [2,2,3]`)

Per copiare bisogna usare lo **slicing** → `a = [1,2,3]; b = a[:]; b[0] = 2 → b è [2,2,3], a è [1,2,3]`

Liste – Operazioni

Le tuple sono **immutabili** (come le stringhe). Si possono usare tutti gli operatori delle liste tranne quelli di modifica.

⇒ `tup1 = ('one', 'two', 12, 25)`

Set

Il set è un insieme non ordinato di oggetti non replicati

- ⇒ Non ordinato → non posso accedere tramite indice
- ⇒ Non replicati → lo stesso oggetto sarà presente al massimo una volta

Creato con la funzione `set()` o `{elem, elem}`

- `a = set()` → insieme vuoto
- `b = set([lista])` → insieme creato dalla lista `lista`
- `c = {1,2}` → insieme con al suo interno gli interi 1 e 2

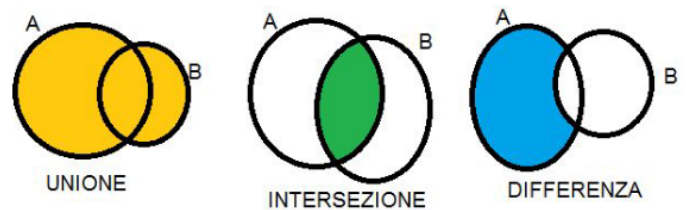
I set sono comodi per certe operazioni:

- Eliminare i duplicati (per es. partendo da una lista)
- Test di appartenenza → più efficiente che scorrere una lista | usando l'operatore **in** come per le liste (nonostante la sintassi/semantica sia la stessa, l'implementazione della ricerca su set è più efficiente che su lista)

Operazioni

- `len(S)` → cardinalità del set `S`

- $x \in S, x \notin S \rightarrow$ appartenenza all'insieme
- $S1.isdisjoint(S2) \rightarrow$ disgiunzione
- $S1.union(S2) \rightarrow$ unione ("|")
- $S1.intersection(S2) \rightarrow$ intersezione ("&")
- $S1.difference(S2) \rightarrow$ differenza ("-")



Dictionary

Il dizionario è un'associazione **key** – **value** di più elementi (simili alle *HashMap* in Java).

Creato con {} o **dict()**

- $a = \{\}$, $a = dict() \rightarrow$ dizionario vuoto
- $b = \{'chiave':valore, 'chiave2':valore2\} \rightarrow$ insieme con due coppie chiave-valore

I valori possono essere qualsiasi oggetto (interi, stringhe, ...).

Le chiavi possono essere solo oggetti **immutabili** (Numerics, Stringhe, Tuple) – esse sono **uniche** nel dizionario.

Operazioni

- $a['chiave'] \rightarrow$ accesso ai valori tramite chiave (lettura, modifica, inserimento)

Metodi

- $.keys() \rightarrow$ restituisce una lista con tutte le chiavi contenute nel dizionario
- $.values() \rightarrow$ restituisce una lista con tutti i valori contenuti del dizionario
- $.items() \rightarrow$ restituisce una lista di tuple (chiave, valore)

Eliminazione

- $del a[chiave] \rightarrow$ elimina la chiave e il valore dal dizionario
- $.pop(chiave) \rightarrow$ elimina e restituisce il valore

Controllo presenza chiave

- Operatore **in** \rightarrow True se la chiave è presente nel dizionario (es. *'uno' in a*)
- $.has_key(chiave) \rightarrow$ True se la chiave è presente nel dizionario

Costrutti di base

Un blocco di codice inizia con i : (seguito dal ritorno a capo) ed è indentato dall'inizio alla fine.

\Rightarrow La fine dell'indentazione indica la fine del blocco (in altri linguaggi, si usano le {})

Non si usa il ; per indicare la fine di un'istruzione, bensì il **ritorno a capo**.

\Rightarrow Si può usare per mettere più istruzioni sulla stessa riga (es. $b = 0$; $b = b + 1$)

È possibile utilizzare la , per assegnamenti multipli (es. $b, a = 1, 2$).

Si commenta riga per riga utilizzando # (non vi è il commento per blocchi).

while b<0:

$b = b+1$

$b = b+2$

b = 0

Moduli e import

È possibile importare moduli esterni (librerie) per poterle usare all'interno del proprio codice.

| | | |
|---|---|---|
| <pre>import pandas as pd a = pd.DataFrame()</pre> <p>Si rinomina il modulo per comodità o per evitare conflitti</p> | <pre>from pandas import DataFrame a = DataFrame()</pre> <p>Si importa una funzione specifica e la si utilizza senza inserire il riferimento al modulo</p> | <pre>from pandas import * a = DataFrame()</pre> <p>Si importano tutte le funzioni del modulo (uso senza rif.)</p> |
|---|---|---|

Costrutti condizionali

| | |
|---|---|
| <pre>if condizione1: codice elif condizione2: codice else: codice</pre> | <p>Tipico if, elif, else</p> |
|---|---|

| | |
|---|--|
| <pre>while b < 10: b = b + 1 else: b = 100</pre> | Ciclo che esegue finchè la condizione è vera, quando esce esegue l' <i>else</i> (se c'è) |
| <pre>for a in elenco: print(a) else: print('vuoto')</pre> | Ciclo che esegue su tutti gli elementi nell'elenco, quando esce esegue l' <i>else</i> (se c'è) |

Comandi di salto

- *break* → interrompe un ciclo for/while
- *continue* → salta all'iterazione for/while successiva
- *else* → può essere inserita alla fine di un blocco relativo ad un ciclo.
 - Viene eseguita (una volta sola) se un ciclo termina le sue iterazioni o quando la condizione di ciclo è valutata a False. **Non viene eseguita in caso di *break*.**

For

Il ciclo for può iterare su qualsiasi elenco di oggetti (stringhe, tuple, liste, ...).

In base al tipo di elenco gli elementi su cui si itera saranno differenti.

| | |
|--|---|
| <p>Un tipico uso è quello di iterare su interi ordinati → <code>for(i = 0; i < 10; i ++).</code></p> <p>Si usa la funzione <i>range</i>(10): <code>for i in range(10):</code> <code> print(i)</code></p> | <pre>range(fine) range(inizio, fine) range(inizio, fine, intervallo) Di default, intervallo = 1</pre> |
| <p>Ciclo for su stringhe: ad ogni iterazione vi è il riferimento ad un carattere Ciclo for su liste/tuple: ad ogni iterazione vi è il riferimento ad un oggetto della lista/tupla Ciclo for su set: ad ogni iterazione vi è il riferimento ad un elemento dell'insieme (NB: non è garantito l'ordine) Ciclo for su dictionary: ad ogni iterazione vi è il riferimento ad una chiave dell'insieme (NB: non è garantito l'ordine)</p> | |
| <p>La funzione <i>enumerate</i> enumera la sequenza su cui si itera, ritornando una tupla (indice, oggetto). <u>Esempio:</u> <code>a = 'abc'</code> ⇒ <code>for i in enumerate(a): print(i)</code> → (0,'a') (1,'b') (2,'c') ⇒ <code>for i,o in enumerate(a): print(i)</code> → 0 1 2</p> | |

Possiamo iterare su molti tipi di elenchi, l'importante è che siano iterabili.

- ⇒ L'elenco deve avere un metodo `__iter__()` che ritorna un iterabile (**duck typing**).
- ⇒ Un oggetto per essere iterabile deve avere un metodo `__next__()` che ritorna l'elemento successivo dell'insieme
- Lo statement *for* chiama `__iter__()` e poi `__next__()` sull'oggetto ritornato

```
a = [1,2]
i = a.__iter__()
i.__next__() -> 1
i.__next__() -> 2
i.__next__() -> Eccezione StopIteration
```

Il *for* cattura questa eccezione per uscire dal ciclo

Ci sono wrapper più comodi per `__iter__()` e `__next__()`: ***iter(a)*** → ***i*** e ***next(i)***.

Queste funzioni semplicemente chiamano i rispettivi metodi sull'oggetto passato come argomento.

Funzioni

Le funzioni sono blocchi di codice (indentato) che non sono innestate in una classe (=> metodi) e si definiscono con ***def***.

- ⇒ Non si specificano i tipi degli argomenti e dei valori ritornati dalla funzione
- ⇒ È possibile definire funzioni con parametri opzionali che assumono un valore di default se non passati durante l'invocazione. Si specifica il valore di default usando `=`. I parametri opzionali devono essere gli ultimi.

Passaggio di argomenti (al richiamo di una funzione)

Normalmente si usa la notazione posizionale (**positional arguments**), ma è possibile specificare a quale parametro ci si riferisce (**keyword arguments**). Si possono mischiare, ma i positional arguments vanno prima dei keyword arguments.

| | | |
|---|--|---|
| <pre>a = funzione(0,1) def funzione(arg1, arg2=0): print(arg1) print(arg2) return 1</pre> | <pre>a = funzione(arg2=3, arg1=1) def funzione(arg1, arg2=0): print(arg1) print(arg2) return 1</pre> | <pre>a = funzione(3, arg2=1) def funzione(arg1, arg2=0): print(arg1) print(arg2) return 1</pre> |
|---|--|---|

È possibile 'spacchettare' sequenze (es. liste) e passare ogni elemento come positional argument.

Si usa l'operatore * **anteposto alla sequenza** $\Rightarrow a = [1, 2, 3, 4, 5] \rightarrow \text{funzione}(*a)$ equivale a $\text{funzione}(1, 2, 3, 4, 5)$

➤ Il numero di parametri deve essere lo stesso di quelli definiti nella funzione (al netto di quelli opzionali)

È possibile 'spacchettare' dizionari e passare ogni elemento come keyword argument.

Si usa ** **anteposto al dizionario** $\Rightarrow a = \{a:1, b:2, c:3\} \rightarrow \text{funzione}(**a)$ equivale a $\text{funzione}(a=1, b=2, c=3)$

Parametri formali con * e **

Gli operatori * e ** possono anche essere utilizzati nella definizione della funzione $\Rightarrow \text{def funzione}(*args, **kwargs)$

In questo caso, tutti i parametri posizionali passati alla funzione vengono condensati nella tupla **args** e tutti i keyword arguments vengono condensati in un dizionario **kwargs** (utile se non si sa a priori il numero di parametri necessari).

➤ La funzione può comunque avere parametri espliciti $\Rightarrow \text{def funzione}(arg1, arg2, *args, **kwargs)$ (che non saranno presenti in **args** e **kwargs**)

Ricorsione

Funzione che richiama se stessa fino a raggiungere una **condizione di stop** (necessaria).

Esempio - Fibonacci iterativo \rightarrow ricorsivo

| | | |
|---|-------------------|---|
| <pre>def fib(n): a, b = 0, 1 print(a) for i in range(n): print(b) a, b = b, a + b</pre> | \longrightarrow | <pre>def fib(n): if (n == 0): print(0) return 0, 1 a, b = fib(n-1) print(a+b)</pre> |
|---|-------------------|---|

Esempio – appiattare una lista di liste

Ci aiutiamo con la funzione **hasattr**

```
def flatten(l):
    res = []
    if not hasattr(l, '__iter__'):
        return [l]
    for o in l:
        res.extend(flatten(o))
    return res
```

"Appiattare una lista di liste" significa trasformare una lista che contiene altre liste annidate in una lista piatta, ovvero in una sequenza lineare in cui tutte le sottoliste sono state eliminate e i loro elementi sono stati spostati nel livello superiore.

Es. $l = [1, [2, [3, 4], 5], 6]$ diventerebbe $[1, 2, 3, 4, 5, 6]$

Programmazione di ordine superiore

Si definisce un linguaggio come **di ordine superiore** se consente di utilizzare funzioni come valori.

⇒ Le funzioni possono essere passate ad altre funzioni e possono essere ritornate come risultato di altre funzioni

```
def print_function_result(func):
    print(f'il risultato è {func()}')

def get_print_help_function(lang):
    def eng_help():
        print('help')
    def ita_help():
        print('aiuto')
    if lang == 'eng':
        return eng_help
    else:
        return ita_help
```

Lambda

Prendiamo in esempio la funzione **map()**, che

⇒ Applica tale funzione a ogni oggetto e

In certi casi la funzione da applicare è molto semplice. Python permette di passare la funzione senza

⇒ Viene detta **lambda function** (o funzione lambda)

⇒ Sintassi: **lambda** <argomenti>:<valore>

⇒ Può contenere un'unica istruzione che

```
def quad(a)
    return a**2
```

————→ **map(lambda x:x**2, [1,2,3])** —————

map(quad, [1,2,3])

CLOSURE

Le funzioni innestate possono accedere alle variabili delle funzioni madre. Tali variabili vengono incapsulate nelle funzioni figlie in modo che possano essere utilizzate anche quando la funzione madre termina.

Generatori

Abbiamo visto che si può iterare su qualsiasi oggetto che abbia un metodo `__iter__` che ritorni un iteratore. Questo implica la costruzione di una classe con tale metodo. I **Generatori** semplificano questo lavoro.

⇒ Possono essere sia funzioni sia espressioni – in entrambi i casi, possono essere utilizzati in un ciclo.

Funzioni generatrici

Esse generano ad ogni iterazione l'elemento successivo – alla fine, è come se ritornassero una serie di valori.

Si creano attraverso il comando **yield** seguito dal valore da ritornare.

⇒ Quando una funzione contiene il comando **yield**, essa viene trattata come un Generatore ed implicitamente dotata dei metodi `__iter__` e `__next__`

Esempio

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

for y in range(10): ...

Noi richiamiamo la funzione, che inizia quindi la sua esecuzione → $i = 0$

Inizia il ciclo while, $0 < 10$ quindi esegue il comando **yield i**, che ritorna il valore i (in questo caso, il primo) e interrompe l'esecuzione della funzione.

y assume tale valore, viene utilizzato e termina la 1° iterazione del ciclo for.

Inizia la 2° iterazione del ciclo for e viene ripresa l'esecuzione della funzione da dove si era interrotta, eseguendo quindi $i = i + 1$ e riprendendo il ciclo (se $i < stop$), e così via....

Alla fine di tutto viene sollevata l'eccezione **StopIteration** per fermare il ciclo while.

Espressioni generatrici

I Generatori possono anche essere creati con una sintassi molto stringata → (espressione **for** variabili **in** sequenza)

⇒ Es. $(x**2 \text{ for } x \text{ in range}(10))$ → ritornerà ad ogni iterazione il quadrato dei numeri da 0 a 9

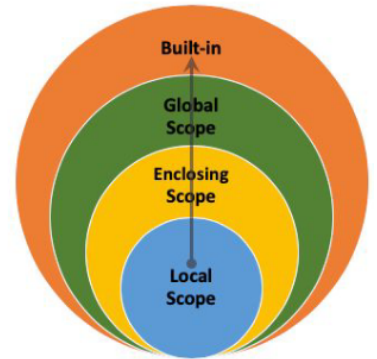
List comprehension

Una sintassi simile (usando `[]` invece che `()`) può essere usata per generare liste in modo immediato

⇒ Es. `a = [x**2 for x in range(10)]` → `a` conterrà una lista dei quadrati da 0 a 9

Scope

- **Built-in** → entità sempre disponibili (es. `print`, disponibile non appena apriamo la shell)
- **Global** → entità definite nel codice (non dentro a funzioni o blocchi)
- **Enclosing** → entità incapsulate per effetto della closure
- **Local** → entità definite dentro il blocco di codice in cui si è



Esempio

```
print('ciao')

a = 1

def encaps():
    b = 2
    def local():
        c = 3 + b

def another():
    d = 0
```

Meccanismo di risoluzione

Si parte dal Local, poi si guarda Enclosed, poi Global e infine Built-in (dal più interno al più esterno).

Questo fa sì che si possono chiamare le variabili con lo stesso nome “nascondendo” quelle degli scope più esterni.

Esempio

```
print('ciao')

x = 1

def encaps():
    x = 2
    def local():
        x = 3

def another():
    x = 0
```

Si può forzare l'utilizzo di un'entità dello scope globale con la parola chiave **global**.

```
print('ciao')

x = 1

def encaps():
    x = 2
    def local():
        global x
        x = 3

def another():
    x = 0
```

Argomenti funzioni

Gli argomenti delle funzioni entrano nel Local scope.

| | |
|---|---|
| <pre>def fun(x): print(x) x = 'ciao' print(x) x = 5 fun(x) print(x)</pre> | <p><u>Output:</u> 5 \n ciao \n 5</p> <p>All'interno della funzione viene cambiata la <code>x</code> locale, ma quella al di fuori rimane la <code>x</code> di prima. È come se la <code>x</code> sia copiata nel parametro della funzione (= passaggio per valore).</p> <p>In realtà, in Python tutto è un oggetto e, nel passaggio degli argomenti, viene copiato il riferimento all'oggetto.</p> <p>Riassegnando <code>x</code> si cambia il riferimento della <code>x</code> locale che ora punterà al nuovo oggetto (<code>x = 3</code>). Tornati nel main, la <code>x</code> locale non esisterà più (Local scope).</p> |
| <pre>def fun(x): print(x) x.append(3) print(x) x = [1,2] fun(x) print(x)</pre> | <p><u>Output:</u> [1,2] \n [1,2,3] \n [1,2,3]</p> <p>In questo caso, è come se fosse stato passato per riferimento.</p> <p>Infatti, in Python viene copiato il riferimento all'oggetto.</p> <p>Se si chiama append lo si chiama sullo stesso oggetto perché tutte e due le <code>x</code> puntano al medesimo oggetto.</p> <p>Nel caso precedente, il riferimento della <code>x</code> locale è stato cambiato con la riassegnazione.</p> |

Gestione delle eccezioni

Nell'ambiente di runtime (RE), viene definita una classe madre che rappresenta eccezioni software generiche.

Ogni anomalia a runtime è associata a una sottoclasse specifica della classe madre.

Le eccezioni possono essere **sollevate** in due modi:

1. Automaticamente: si verificano in seguito a anomalie a runtime causate da istruzioni (es. divisione per zero)
2. Manualmente: sollevate manualmente utilizzando istruzioni come **"raise"** (Python) o **"throw"** (Java)

In caso di eccezione, viene eseguito del codice di gestione. Se non è presente un gestore specifico, verrà utilizzato un gestore predefinito che stamperà lo stack delle chiamate ed uscirà dal programma.

| | |
|--|---|
| Sintassi per <u>catturare</u> un'eccezione | Spiegazione: |
| <pre>1 try: codice 2 except Classe_eccezione as identificativo: codice 3 except Altraclasses_eccezione as identificativo: codice 4 except: codice 5 else: codice 6 finally: codice</pre> | <ol style="list-style-type: none">1. Viene eseguito il codice nel <i>try</i>.2. Viene eseguito se viene sollevata un'eccezione del tipo <i>Classe_eccezione</i>3. Stessa cosa di prima ma per <i>Altraclasses_eccezione</i>4. Viene eseguito se viene sollevata un'eccezione non catturata esplicitamente prima5. Viene eseguito dopo il <i>try</i>, solo se si conclude senza aver sollevato eccezioni6. Viene eseguito dopo il <i>try</i>, a prescindere che abbia sollevato o meno eccezioni <p>NB: <i>try, except ..., except, else, finally</i> (in quest'ordine)</p> <p>Ci possono essere più coppie <i>Classe_eccezione/identificativo</i> se si vuole utilizzare lo stesso codice per gestirle.</p> |
| Sintassi per <u>sollevare</u> un'eccezione raise NameException(arg1,arg2) | |

Decoratori

Si usano per aggiungere funzionalità ad una funzione (es. misurare il tempo di esecuzione di una funzione)

Si definisce una funzione che prenda come parametro una funzione e restituisca una funzione.

⇒ Funziona indipendentemente dal numero di parametri che la funzione richiede.

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

⇒ Possiamo applicare il decoratore a tutte le funzioni che vogliamo con la sintassi **@**

Classi

Una **classe** definisce variabili e metodi contenuti, un **oggetto** è l'istanza di una classe.

Ricordiamo i 4 pilastri della OOP (Object-Oriented Programming):

1. Astrazione → non si guardano i dettagli implementativi (per l'utilizzatore)
2. Incapsulamento → protezione degli attributi interni
3. Ereditarietà → si ereditano comportamenti (meno codice)
4. Polimorfismo → comportamento adattabile

| Definizione | Istanza |
|---|--|
| <pre>class nome_classe: attributo1 = valore1 def metodo1(self, arg1, arg2): ...</pre> | <p>Utilizzando l'oggetto classe è possibile istanziare oggetti di quella classe</p> <pre>variabile = NomeClasse()</pre> <p>Accesso ai contenuti della classe (attributi di classe e metodi):</p> <pre>NomeClasse.nome_elemento</pre> |

Attributi di classe

Attributi che appartengono all'oggetto classe (e non agli oggetti istanziati di quella classe).
Possono essere acceduti direttamente dall'oggetto classe o utilizzando gli oggetti istanziati:

```
NomeClasse.i
a = NomeClasse()
a.i
```

NB: se assegno usando l'oggetto $\rightarrow a.i = 20 \rightarrow$ ora l'oggetto ha un suo attributo i che nasconde quello di classe.

Metodi

Si definiscono con la parola chiave **def** e hanno come primo argomento obbligatorio il riferimento all'oggetto stesso.

- ⇒ Tipicamente lo si chiama **self** e permette di accedere ai metodi/attributi dell'oggetto (istanziato).
- Quando si chiama il metodo su un oggetto **non** si passa **self** – siccome il metodo è chiamato su un oggetto specifico, l'interprete è a conoscenza che **self** sarà riferito a quell'oggetto.

Costruttore

Metodo speciale che inizializza l'oggetto istanziato e deve avere come nome **__init__**.

Al suo interno si inizializzano gli attributi dell'oggetto (dell'istanza, non quelli di classe) utilizzando **self**.

```
class NomeClasse():
    def __init__(self, arg1, arg2):
        self.a = 10
```

Il costruttore viene chiamato automaticamente quando viene istanziato l'oggetto $\rightarrow a = \text{NomeClasse}()$

Metodi static e class

Metodi che possono essere chiamati sulla classe stessa invece che sull'istanza.

- ⇒ Si sfruttano i decorator **@staticmethod** e **@classmethod**.
- ⇒ Potranno accedere solo agli attributi di classe (e non quelli di istanza)
- I metodi statici non richiedono il primo parametro **self** obbligatorio (nessuna istanza considerata)
- I metodi di classe richiedono il primo parametro **self** che sarà la l'oggetto classe stesso (e non un'istanza)

Esempio

```
class Date(object):
    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        date1 = cls(day, month, year)
        return date1

    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999
```

Ciclo di inizializzazione

Quando si istanzia un oggetto, l'interprete chiama due metodi: il 1° per creare l'oggetto, il 2° per inizializzarlo.

- ⇒ L'inizializzazione è il costruttore `__init__` visto prima – non restituisce nulla in quanto l'oggetto esiste già (è *self*) e viene solamente inizializzato.
- ⇒ La creazione è il metodo `__new__` che tipicamente non viene definito in quanto si usa quello di default. Prende in ingresso una classe e si occupa di restituire un oggetto di quella classe (non ancora inizializzato)

Esempio

```
class class_name:
    def __new__(cls, *args, **kwargs):
        print( "ciao io sono __new__" )
        return object.__new__(cls)
```

Restituisce l'oggetto che verrà poi passato ad `__init__`.
In questo esempio, si chiama `__new__` della classe madre *object*.

Metodi/attributi privati

In Python, i metodi e gli attributi di una classe sono pubblici.

Per rendere privata un'entità si agisce sul nome → entità che iniziano con (2) `__` e terminano con massimo un `_`

Esempio

```
class Prova:
    def __init__(self):
        self.__priv = 'privato'
```

Python le nasconde rinominandole come `_NomeClasse__NomeEntità` così da impedire l'accesso con `.NomeEntità`, ma in realtà è comunque possibile accederle dall'esterno.

Metodi speciali

Iniziano e terminano con (2) `__` → possono essere definiti e hanno una semantica speciale.

- ⇒ Vengono chiamati in automatico dall'interprete in certe situazioni (es. `__init__` per inizializzare un oggetto)

- `__init__` costruttore
- `__repr__` rappresentazione dell'oggetto
- `__getattr__` per emulare l'accesso ad un attributo: `a.x --> a.__getattr__('x')`
- `__getitem__` per emulare un tipo contenitore (tipo le liste): `a[x] -> a.__getitem__('x')`
- `__add__`, `__mul__`, etc. per emulare somme, moltiplicazioni etc.
- `__call__` l'oggetto può essere chiamato come una funzione
- `__str__` converte un oggetto in una stringa
- `__del__` distrugge la classe (metodo distruttore)
- `__{ eq , gt , ge , lt , le }__` verifica se un valore è { uguale, ..., ... } ad un altro
- `__setitem__` operatore `[]` in uscita (assegnazione)

Notare che questi metodi sono associati ad operatori Python (`[]`, `+`, `*`, ...) → questo significa che, quando si utilizzano questi operatori, in realtà vengono chiamati questi metodi. È possibile avere comportamenti specializzati per i vari operatori in base a quale oggetto è coinvolto.

Polimorfismo attraverso l'overloading

Sovrascrivere i metodi speciali di default può essere molto utile per personalizzare il comportamento di un oggetto.

- ⇒ `__getattr__` e `__setattr__` vengono utilizzati per accedere e settare gli attributi di un oggetto.
Sono i veri metodi utilizzati quando si usa l'operatore `.` e `=`
- Sovrascriverli può avere molte finalità:

Aggiornare un altro campo in conseguenza di un settaggio

```
class Prova:
```

```
    def __setattr__(self, campo, valore):  
        if campo == 'saldo':  
            super().__setattr__(campo, valore) ← NON USARE self.__setattr__(campo, valore)  
            self.movimenti.append(valore)          RICORSIONE!
```

Proteggere l'accesso

```
class Prova:
```

```
    def __getattr__(self, campo):  
        if campo == 'saldo':  
            raise AttributeError('campo non accessibile')  
        else:  
            return super().__getattr__(campo)
```

```
a = Prova()
```

```
a.saldo -> Error!
```

Getters and Setters

Tipicamente, nei linguaggi ad oggetti si proteggono i dati dell'oggetto e ci si accede tramite i metodi **getters** e **setters**. In Python, come già visto, è possibile proteggere (o quasi) un attributo dall'accesso esterno.

Definendo metodi per l'accesso

```
class Prova:
```

```
    def __init__(self, x):  
        self.__x__ = x  
    def getx(self):  
        return self.__x__  
    def setx(self, x):  
        self.x = __x__
```

Usando `__getattr__`

```
class Prova:
```

```
    def __init__(self, x):  
        self.__x__ = x  
    def __getattr__(self, attr):  
        if attr == 'x':  
            return self.__x__  
        else:  
            return super().__getattr__(attr)  
    def __setattr__(self, attr, value):  
        if attr == 'x':  
            self.__x__ = value  
        else:  
            super().__setattr__(attr, value)
```

NB: se ho molti attributi, l'*if* diventa ingestibile

Il decoratore `@property` è un decoratore built-in che permette di annotare quelli che sono i getters e i setters per un attributo (potrebbe anche **non essere in memoria**).

- ⇒ Con `@property` si annota il metodo che deve restituire il valore. Lo si accederà attraverso il nome del metodo (che però verrà trattato come attributo)
- ⇒ Con `@<nome>.setter` si annota il setter dell'attributo

Usando **@property**

class Prova:

```
def __init__(self, x):
```

```
    self.__x_ = x
```

```
@property
```

```
def x(self):
```

```
    return self.__x_
```

```
@x.setter
```

```
def x(self, new_x):
```

```
    self.__x_ = new_x
```

Es. attributo non in memoria

Ereditarietà

Permette di far ereditare i metodi e gli attributi di una classe madre alle classi figlie. Python permette quella **multipla**.

| Esempio | | Spiegazioni |
|-----------------------|--------------------------------|--|
| class Automobile: | class Fuoristrada(Automobile): | <i>Fuoristrada</i> eredita tutti i metodi/attributi di <i>Automobile</i> . ⇒ La classe madre viene esplicitata tra parentesi ⇒ I metodi possono essere sovrascritti (es. <i>muoviti</i>) ⇒ Si usa <i>super()</i> per chiamare un metodo della classe madre |
| def __init__(self): | def ridotte(self): | |
| self.posizione = 0 | self.rapporto = 0.01 | |
| def muoviti(self, l): | def muoviti(self, l): | |

```
    self.posizione += l
```

```
    l = l*self.rapporto
```

```
    super().muoviti(l)
```

Ereditarietà multipla

| Esempio | | Spiegazioni |
|-----------------------|-----------------------|--|
| class Automobile: | class Barca: | <i>Anfibio</i> eredita da entrambi le classi ⇒ Per decidere quale <i>muoviti</i> verrà utilizzato, Python parte dalla classe figlia – se il metodo non è definito lì, passa alle classi madri da <u>sinistra</u> a <u>destra</u> . ⇒ Si può verificare l'ordine di risoluzione (Method Resolution Order) accedendo all'attributo <code>__mro__</code> (in sola lettura) o usando il metodo <i>mro()</i> |
| def __init__(self): | def affonda(self): | |
| self.posizione = 0 | self.affondato = True | |
| def muoviti(self, l): | | |

```
    self.posizione += l
```

`__bases__` è un attributo che indica le classi base (madri) e può essere modificato (in tal caso, `__mro__` viene ricalcolato).

```
a = Anfibio()
print(Anfibio.__mro__)
Anfibio.__bases__ = (Automobile, Barca)
print(Anfibio.__mro__)
```

Se non specificato di default nella classe figlia, di default viene invocato il **costruttore** della prima classe madre. Per invocarli entrambi bisogna esplicitarlo:

```
class Anfibio(Barca, Automobile):
```

```
def __init__(self):
```

```
    Barca.__init__(self)
```

```
    Automobile.__init__(self)
```

Check classe

⇒ *isinstance*(*ist*, *classe*) → serve per verificare il tipo di un'istanza di una classe (es. *isinstance*(*x*, *int*))

⇒ *issubclass*(*x*, *y*) → serve per verificare se *x* è una sottoclasse di *y*

Accesso agli attributi

Un oggetto può avere più attributi: *self*, di classe, ereditati, metodi di accesso "fake" agli attributi (*__getattr__*).

➤ Alcuni possono sovrascrivere od oscurare altri.

Come capire a quale attributo si sta accedendo? Prima i metodi o gli attributi?

⇒ A livello di eredità fa fede l'attributi del livello più basso (come in altri linguaggi). Quando si accede un attributo si guarda prima se vi è nelle classe figlia – se non vi è, si passa alla classe madre (attributi di classe).

Per gli attributi di istanza dipende da cosa fanno i vari *__init__* e da come vengono chiamati.

| | | | |
|---|--|---|--|
| <pre>class Animale: x = 0 y = 0 def __init__(self): self.a = 1 self.b = 1</pre> | <pre>class Cane(Animale): x = 10 def __init__(self): self.a = 2 super().__init__()</pre> | <pre>class Animale: x = 0 y = 0 def __init__(self): self.a = 1 self.b = 1</pre> | <pre>class Cane(Animale): x = 10 def __init__(self): super().__init__() self.a = 2</pre> |
| <pre>class Animale: x = 0 y = 0 def __init__(self): self.a = 1 self.b = 1</pre> | <pre>class Cane(Animale): x = 10 def __init__(self): self.a = 2</pre> | <pre>class Animale: x = 0 y = 0 def __init__(self): self.x = 1 self.b = 1</pre> | |

In realtà, la documentazione dice: "un'istanza di classe ha un namespace implementato come dizionario che è il primo posto in cui vengono cercati i riferimenti agli attributi. Quando un attributo non viene trovato lì e la classe dell'istanza ha un attributo con quel nome, la ricerca continua con gli attributi della classe".

⇒ Questo namespace è l'attributo speciale *__dict__*

a = Animale()

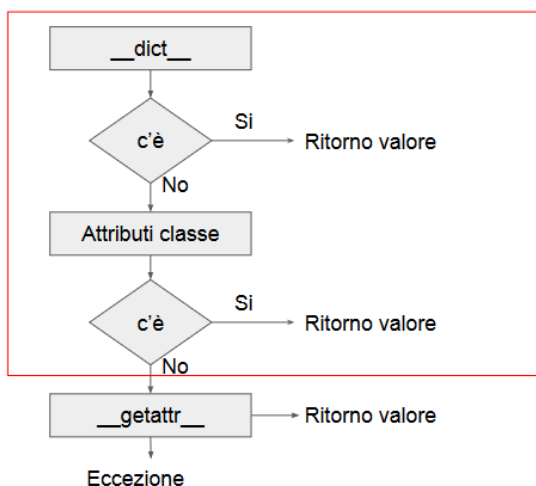
⇒ Ci sono tutti gli attributi di istanza

a.a = 1

➤ Ogni volta che faccio *self.x = <valore>* (anche nell'*__init__*) esso viene inserito nel *__dict__* *a.__dict__*

Se un attributo non viene trovato né in *__dict__* né tra gli attributi di classe si passa al metodo speciale

__getattr__



__getattr__ è essere ridefinito dall'utente

Questo esempio funzionava perché *saldo* non è né un attributo di classe né inizializzato in *__init__*

```
class Prova:
    def __getattr__(self, campo):
        if campo == 'saldo':
            raise AttributeError('campo non accessibile')
        else:
            return super().__getattr__(campo)
```

| | |
|--|--|
| <p>Se volessi avere veramente il campo, non si arriverebbe mai all'esecuzione di <code>__getattr__</code></p> <pre> class Prova: def __init__(self): self.saldo = 0 def __getattr__(self, campo): if campo == 'saldo': raise AttributeError('campo non accessibile') else: return super().__getattr__(campo) </pre> | <p>In realtà si passa per il metodo <code>__getattribute__</code> che di fatto controlla nel <code>__dict__</code> e negli attributi di classe (può essere personalizzato) – dovrei quindi usare questo</p> <pre> class Prova: def __init__(self): self.saldo = 0 def __getattribute__(self, campo): if campo == 'saldo': raise AttributeError('campo non accessibile') else: return super().__getattribute__(campo) </pre> |
|--|--|

`__getattribute__` viene sempre invocato quando si accede ad un attributo, tranne in casi speciali in cui vengono chiamati dall'interprete gli attributi speciali (es. nei cicli `__iter__` e `__next__`: se usiamo `len()` che chiama `__len__`).

Esempio – se usiamo `len()` che chiama `__len__`

`len()` restituisce la lunghezza di un elemento (es. elementi in una lista) e, sotto l'interprete, chiama `__len__`

```

class MyList(list):
    def __getattribute__(self, item):
        print(f'getattribute {item}')
        return super().__getattribute__(item)

    def foo(self):
        print('mi hai chiamato')
        pass

```

`__setattr__`

Di fatto, modifica (o aggiunge) la corrispettiva entry in `__dict__`.

- Non vi è il corrispettivo di `__getattribute__` (non vi è un `__setattr__`).
- Viene invocato ogni volta che si fa un assegnamento (anche nell'`__init__`)
(Es. `self.x = 10` → `self.__setattr__('x', 10)` - ecco perchè non si modifica l'attributo di classe)

Esempio

```

class Prova:
    def __init__(self):
        self.saldo = 0

    def __setattr__(self, campo, valore):
        if campo == 'saldo':
            raise AttributeError('campo non modificabile direttamente')
        else:
            return super().__setattr__(campo, valore)

```

Problema

Anche nell'`__init__` verrà invocato il `__setattr__` impedendone l'inizializzazione.

Soluzione

Accedendo tramite `__dict__` si **bypassa** la `__setattr__` →
`self.__dict__['saldo'] = 0`

@property

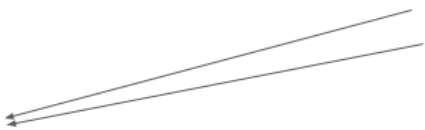
Ricordiamo che è un decoratore. È un attributo di istanza o di classe o un metodo? In che punto viene controllato? Le funzioni sono oggetti in Python. Il decoratore può prendere in ingresso qualsiasi entità e restituire qualsiasi entità. In Python un **attributo** può essere anche un'entità definita come **descrittore**. In pratica, un descrittore è un attributo il cui valore è un oggetto che implementa uno o più dei seguenti metodi: `__get__`, `__set__`, `__delete__`

⇒ In questo caso, l'interprete non setta/accede/cancella l'attributo direttamente ma chiama il corrispettivo metodo

| Esempio | |
|---|--|
| <pre>class Desc: def __get__(self, istanza, owner): pass class Prova: i = Desc()</pre> | <pre>p = Prova() p.i -> viene chiamato __get__ di Desc __get__ in Desc: ⇒ self: istanza di <i>Desc</i> (nel nostro es. <i>i</i> di <i>Prova</i>) ⇒ istanza: oggetto sul quale viene richiesto l'accesso all'attributo (nel nostro es. <i>p</i>, istanza di <i>Prova</i>) ⇒ owner: classe dell'oggetto sul quale viene richiesto l'accesso all'attributo (nel nostro es. <i>Prova</i>)</pre> |
| <pre>class Desc: def __set__(self, istanza, value): pass</pre> | <pre>__set__ in Desc: ⇒ self, istanza: oggetto <i>Desc</i> e oggetto <i>Prova</i> (come in <code>__get__</code>) ⇒ value: valore da settare</pre> |
| <pre>class Desc: def __delete__(self, istanza): pass</pre> | <pre>__delete__ in Desc: ⇒ Oggetto <i>Desc</i> e oggetto <i>Prova</i> (come in <code>__get__</code>)</pre> |

Mettendo insieme i pezzi:

- ⇒ Se `@property` ritorna un oggetto descrittore, tale oggetto potrà avere una funzione nel `__get__` (la funzione decorata). Tale descrittore verrà associato al nome della funzione originale.
- ⇒ Esso diventerà un attributo di classe in quanto non è inizializzato usando `self`

| | |
|--|---|
| NB: a questo punto l'oggetto sarà il medesimo per tutte le istanze della classe <i>Prova</i> | |
| (a <code>@property</code> non interessa in quanto usa una funzione dell'utente) | |
| <pre>class Desc: def __get__(self, istanza, owner): pass class Prova: i = Desc() class Desc: def __init__(self): self.x = 0 def __set__(self, instance, value): self.x = value def __get__(self, istanza, owner): return self.x class Prova: i = Desc()</pre> | <pre>p1 = Prova() p2 = Prova() p1.i p2.i</pre>  |

Se si vuole che ogni oggetto abbia la sua istanza del valore di *i* bisogna usare ***instance***

```
class Desc:
```

```
    def __init__(self):
        self.x = 0

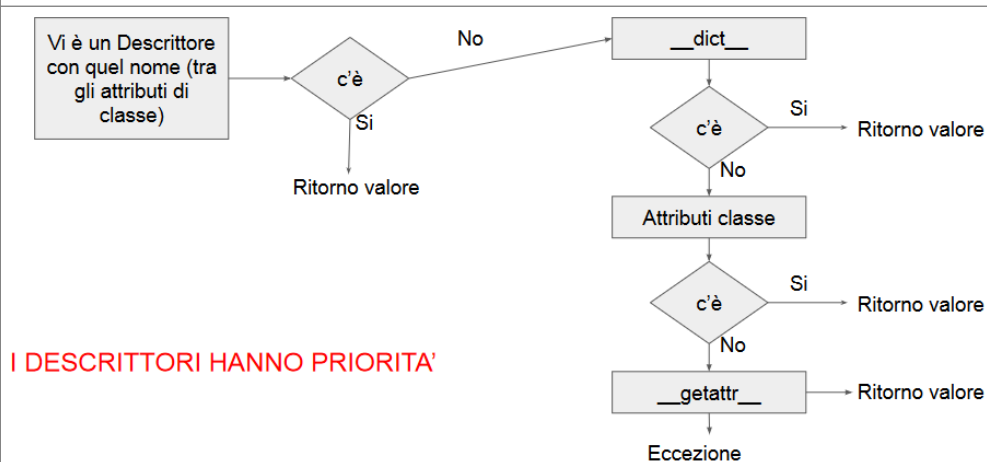
    def __set__(self, instance, value):
        instance.__dict__['x'] = value

    def __get__(self, instance, owner):
        return instance.__dict__['x']
```

```
class Prova:
```

```
    i = Desc()
```

In questo caso si utilizza `__dict__` per evitare ricorsioni, ma questo caso implica che **al 2° accesso non andrei a verificare gli attributi di classe e non vedrei il descrittore**.



Metaclassi

In programmazione a oggetti, una metaclassa è una classe le cui istanze sono a loro volta classi.

In Python tutto è un oggetto → anche le classi sono oggetti → sono quindi istanze di un'altra classe.

➤ In Python, la metaclassa madre è ***type***. Tutte le classi sono istanze di *type*.

Internamente, Python crea le classi (le definizioni delle classi) istanziando la metaclassa *type*.

È possibile estendere *type* e creare una propria metaclassa, dove si ridefiniscono le funzioni `__init__` e la `__new__` in modo da modificare il comportamento di base delle classi.

```
class MiaClasse(metaclass=MiaMetaclasse):
```

```
    pass
```

type() ha due comportamenti:

- Con un argomento ritorna il tipo dell'oggetto;
- Con tre argomenti ritorna una nuova classe (è l'equivalente di scrivere una nuova classe)

```
class X(Y):
```

```
    a = 1
```

Esempio – `M = type('X', (Y), dict(a=1))` da *type*

```
class MiaMetaclasse(type):
```

```
    def __new__(cls, classname, super, classdict):
        return super().__new__(cls, classname, super, classdict)

    def __init__(self):
        super().__init__()
```

Le metaclassi possono essere utili per avere meccanismi simili all'ereditarietà ma anche per creare classi in modo diverso a runtime.

Es. si potrebbero creare classi con attributi di classe che hanno nomi definiti dall'utente

Metodo `__call__`

Metodo speciale che viene chiamato quando un oggetto viene invocato come una funzione

class Prova:

```
def __call__(self):  
    print('eccomi')
```

a = Prova()

a()

Questo implica che anche una classe può essere un decoratore

class dec:

```
def __init__(self, f):  
    self.f = f  
def __call__(self):  
    pass
```

@dec
def foo():
 print('ciao')

⇒ *foo* viene passato ad `__init__`
⇒ Il nuovo oggetto ritornato verrà associato a *foo*

Classi astratte

Una classe astratta è una classe che definisce almeno un metodo astratto.

Un metodo stratto non ha implementazione e va definito nelle sottoclassi.

Python non ha built-in la possibilità di definire metodi astratti, ma c'è un modulo che lo permette (**Abstract Base (ABC)**).

⇒ Al suo interno vi è il decoratore `@abstractmethod` che permette di decorare i metodi che si vogliono come astratti → `from abc import ABC, abstractmethod`
`from abc import ABC, abstractmethod`

```
class Astratta(ABC):  
    @abstractmethod  
    def metodo_astratto(self):  
        pass
```

Deve ereditare da ABC
E solo da altre classi astratte!

Programmazione funzionale

Paradigmi di programmazione:

- Imperativo (es. C)
 - o Il programma è una serie di istruzioni che vengono eseguite una dopo l'altra
 - o Il programmatore scrive le funzioni per risolvere il problema
- Ad oggetti (es. Java)
 - o Il programma è organizzato in oggetti che esibiscono metodi
 - o Gli oggetti incapsulano degli attributi che non vengono esibiti
- Funzionale (es. LISP)
 - o Programmazione in stile matematico
 - o Si dichiara che funzione utilizzare per ottenere un risultato
- Logico (es. Prolog)
 - o Si esprime il problema sotto forma di vincoli
 - o Si deve trovare una soluzione che soddisfi i vincoli

Il **paradigma funzionale** è un paradigma in stile dichiarativo (si dichiara che valore deve assumere un dato nome).

Tipicamente ad un nome viene assegnato il risultato di una funzione – si dice che utilizza “espressioni” come in matematica. Le espressioni vengono valutate per portare al risultato (focus su cosa risolvere).

1. Le funzioni utilizzate in questo paradigma vengono definite **pure**
 - o Stesso input implica stesso risultato
 - o Non modificano gli argomenti in input
2. Non si utilizzano i cicli (ma la **ricorsione**)
3. Sono **linguaggi di ordine superiore** (le funzioni possono essere passate come parametri o ritornate come valore)

4. I **nomi** (che in altri linguaggi chiamiamo variabili) sono **immutabili** (per questo non si chiamano variabili)

In Python è possibile programmare in stile funzionale poiché ogni requisito viene soddisfatto.

1. Per le funzioni pure basta non modificare i parametri in ingresso e al limite copiarli e lavorare sulle copie
2. Se necessario, si usa la ricorsione
3. Le funzioni sono oggetti (quindi passabili come argomenti o ritornabili da altre funzioni)
4. Il programmatore può consapevolmente non modificare una variabile (Python)

Esempio

```
def fattoriale(x):  
    return 1 if x<=1 else return x*fattoriale(x-1)  
  
fact10 = fattoriale(10)  
... da qui in poi non devo cambiare fact10 (immutabile)
```

La ricorsione può saturare lo stack – per questo Python ha un limite di chiamate ricorsive permesse.

Lo si può modificare con `import sys ... sys.setrecursionlimit(<massimo_chiamate>)`

Vantaggi

1. Maggior facilità di debug → il risultato di una funzione è dato solamente dagli argomenti
2. Funzioni tendenzialmente più semplici → assenza di cicli da seguire, assenza di flussi complessi
3. Struttura più interpretabile da un compilatore:
 - a. Maggiori ottimizzazioni
 - b. Possibilità parallelizzazioni automatiche
 - c. **Lazy evaluation** → esecuzione solo quando si ha effettivamente bisogno del risultato

Svantaggi

1. Cambio di paradigma implica cambio di abitudini → mentalità diversa, struttura del codice diversa
2. Poco utilizzata → l'imperativo è più semplice

Strumenti per la programmazione funzionale in Python

List Comprehension

Costrutto sintattico per la creazione di liste (o dizionari) senza un ciclo for classico (ma innestato dentro la lista).

⇒ `[expression for value in iterable if condition]`

Sintassi con stesse parole di un ciclo for ma la semantica è differente → il risultato è una lista risultante da tutti i risultati di *expression* calcolati con ogni *value* nell'*iterable* (se la *condition* è vera).

Esempi:

- ⇒ `[k*k for k in range(1, n+1)]`
- ⇒ `[k for k in range(1, n+1) if n%k == 0]`
- **Set** → `{k*k for k in range(1, n+1)}`
- **Dictionary** → `{k : k*k for k in range(1, n+1)}`

Vi possono essere anche più for: `[x for iter2 in iter1 for x in iter2]`

Notare che i nomi vengono letti da sinistra verso destra → `[x for x in iter2 for iter2 in iter1]` => **ERRORE!**

Lambda

Fondamento della programmazione funzionale. Ogni funzione ha argomenti e risultato (funzioni **pure**).

Potenzialmente un intero programma può essere scritto con sole funzioni lambda (**Turing Completo**).

⇒ *lambda argomenti: espressione*

Questa funzione può avere un numero qualsiasi di argomenti ma solo un'espressione, che viene valutata e restituita.

Esempio `filter()`: filtra in base al risultato di una funzione (*True* tiene, *False* scarta)

=> `filter(lambda x: (x%2 != 0), iter1)`

Esempio `map()`: mappa il risultato di una funzione => `map(lambda x: x*2, iter1)`

Map/Reduce

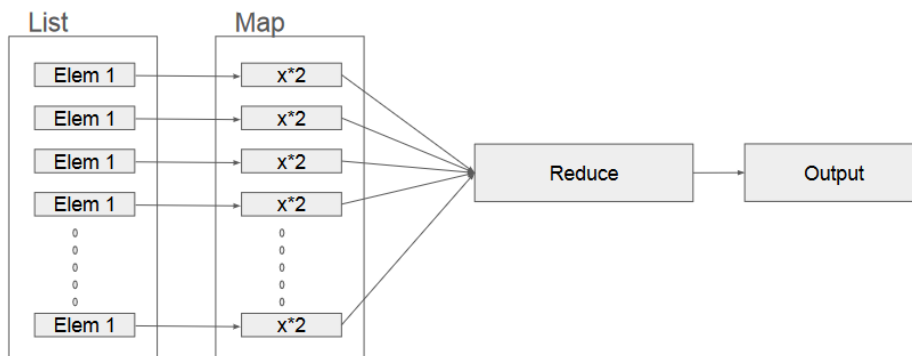
Paradigma distribuito (esiste anche un framework di Google con lo stesso nome) per la computazione dei dati.

Diviso in due fasi:

1. Mappatura dei dati (filtri ed elaborazioni sui singoli elementi)
2. Riduzione (aggregazione dei risultati della fase 1)

Essendo distribuito non si può assumere la sequenza dei dati – la riduzione può avvenire in più step.

Utile in database distribuiti in cui i dati sono in nodi diversi e la computazione non può avvenire in un unico nodo.



In Python esiste la funzione ***reduce()*** che è parte della libreria ***functools***.

- ⇒ Esegue in locale e non in un ambiente distribuito, ma è comunque utile
- ⇒ Applica ripetutamente una funzione di 2 argomenti sugli elementi di una sequenza in modo da ridurre la sequenza ad un unico valore. Più nei dettagli:
 - Ad ogni iterazione il 1° argomento è il risultato delle passate precedenti ed il 2° è un elemento (non ancora processato) della lista.
 - Alla 1° passata si usa il 1° elemento della lista (niente passate precedenti) oppure si può specificare che valore utilizzare come primo risultato (fake).

Esempi:

- ⇒ Somma = > `reduce(lambda x,y: x+y, [1,2,3,4])`
- ⇒ Creazione id un set da una lista = > `reduce(lambda x,y: x.union(set([y])), set(), set())`
 - 1° risultato fake = > `reduce(lambda x,y: x.union(set([y])), [1,2,3,4], set())`

Questo paradigma è efficace sia quando la computazione può essere distribuita, sia quando può essere trattata come un flusso. Questo approccio riduce le risorse necessarie alla computazione.

Eval

Funzione che valuta un'espressione **passata sotto forma di stringa** -> l'interprete la esegue come se fosse codice.

- ⇒ Si possono passare anche variabili dal programma "principale", sia come globali sia come locali al codice della stringa → `eval("print('ciao')", globals, locals)`
 - Esempio = > `a,b = 10,20 eval("print(f'c: {c} d:{d}']", {'c':a}, {'d':b})`
- ⇒ Il suo valore di ritorno è il risultato dell'espressione valutata = > `eval("1+1") → 2`

Può valutare qualsiasi espressione con tutti i problemi relativi alla sicurezza. È buona norma non far valutare stringhe generiche oppure settare accuratamente *globals* e *locals* in modo da non dare accesso a funzioni critiche.

Exec

È molto simile ad *eval* ma più completo: può eseguire anche **istruzioni** (non solo espressioni) e **codice compilato**.

1. Esempio = > `exec("import subprocess")`
2. Ci si riferisce al fatto che anche in Python vi è un formato intermedio (bytecode) = > `exec(compiled)`

Compile

Compila una stringa (o un file) in bytecode => `compile(source, file, mode)`

- `compile()` → compila una stringa (o un file) in bytecode
- ***source*** è la stringa da compilare => `compile("print()", "<string>", mode)`
- ***file*** è il file da compilare => `compile("", "source.py", mode)`
- ***mode*** è la modalità:
 - `'exec'` se contiene delle istruzioni
 - `'eval'` se contiene una singola espressione
 - `'single'` se contiene una singola istruzione interattiva

Functools

Modulo utile per la programmazione funzionale in Python che fornisce:

- ⇒ Funzioni di ordine superiore per eseguire operazioni comuni nella programmazione funzionale
- ⇒ Fornisce anche due classi per rappresentare funzioni: ***partial*** e ***partialmethod***

partial rappresenta un funzione in cui alcuni argomenti sono definiti prima di chiamarla

Quando si chiama *sommapartial* basterà passare solo un argomento.

La *y* è stata definita prima.

partialmethod è simile a *partial* ma lavora sui metodi di una classe.

È possibile definire un *partialmethod* che richiama un altro metodo ma con dei parametri preimpostati.

```
class A:
    def print_str(self, s):
        print(s)
    print_ciao = partialmethod(print_str, s='ciao')
```

cmp_to_key trasforma una funzione di comparazione in una funzione chiave.

Esempio: *sorted()* ordina i valori di una lista e c'è la possibilità di specificare una funzione chiave

⇒ *sorted([(1,2), (2,1)], key = lambda x:x[1])*

In questo caso, ad ogni elemento è applicata la funzione *key*, e poi vengono ordinati usando questo risultato.

NB: non avviene nella funzione *lambda* il confronto tra due elementi (necessario all'ordinamento).

Esempio: supponiamo di avere oggetti più complessi da ordinare

```
class Data:
    def __init__(self, anno, mese, giorno):
        self.anno = anno
    ...
```

Non è banale restituire una chiave che *sorted* possa utilizzare per l'ordinamento, possiamo però scrivere una funzione di confronto tra due date e poi utilizzare *cmp_to_key* per avere una funzione chiave => *cmp_to_key(cmp_date)*

```
def cmp_date(x,y):
    if x.anno < y.anno:
        return -1
    elif x.anno > y.anno:
        return 1
    else: #caso stesso anno
        if x.mese < y.mese:
            return -1
        elif x.mese > y.mese:
            return 1 ...
```

total_ordering è un decoratore per classi che fornisce i metodi di confronto per quella classe.

Basandosi sui metodi *__eq__* e uno tra gli altri metodi di confronto (*__lt__()*, *__le__()*, *__gt__()*, *__ge__()*) inferisce gli altri.

Quindi basterà implementare due metodi per avere gli altri

```
@total_ordering
class A:
    def __eq__(self, other):
        ...
    def __lt__(self, other):
        ...
```

Itertools

Modulo che fornisce funzioni per generare sequenze di dati iterabili (molto utile in programmazione funzionale).

iteratori infiniti

iteratori che modificano la sequenza

`count(10) -> 10,11,12 ...`

`accumulate([1,2,3,4]) -> 1 3 6 10`

`cycle([1,2,3]) -> 1, 2, 3, 1, 2, 3, ...`

`chain('abc', 'def') -> a b c d e f`

`repeat('a') -> a,a,a,a...`

`zip_longest('ABC', 'xyz') -> Ax By Cz`

`starmap(pow, [(1,2), (3,4)]) -> applica pow(*x) -> 1 81`

iteratori combinatori

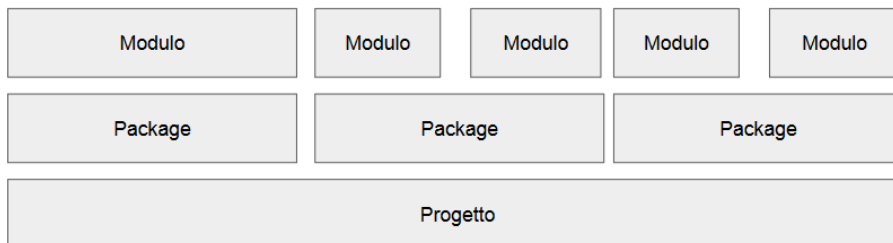
`permutation([1,2,3]) -> tutte le permutazioni possibili`

`combination([1,2,3]) -> tutte le combinazioni possibili`

Struttura progetto

È buona abitudine organizzare un progetto software in più moduli (e file)

- ⇒ Maggior leggibilità e navigabilità del progetto
- ⇒ Maggior riutilizzo di moduli in altri progetti
- ⇒ In certi linguaggi (specialmente quelli compilati) vengono ottimizzati i tempi di compilazione
 - I moduli senza modifiche non vengono ri-compilati



- Un **modulo** è composto da più entità relative a quel modulo, dotato di un fine preciso.
 - In Python, un modulo corrisponde a un file sorgente (entità → funzioni, costanti, classi)
- Un **package** racchiude più moduli inerenti a una stessa area del programma.
 - In Python, è una directory contenente più file (moduli). È possibile organizzarlo in sotto-package.

Per far sì che una directory sia vista come package è necessario inserire al suo interno un file `__init__.py`

Utilizzo di un modulo

Per utilizzare un modulo bisogna importarlo (... o tutte le sue entità, o una specifica) → *import, from...import...*

Per utilizzare un modulo contenuto in un package la sintassi è la stessa → *import <package>.<subp>.<modulo>*

Quando si **importa** un modulo, viene **eseguito** quel file (oltre a definire le entità, è possibile eseguire inizializzazioni).

⇒ **NB:** le entità possono essere accedute

Utilizzo di un package

Anche i package possono essere importati, e la loro inizializzazione può essere messa nel file `__init__.py`

- ⇒ L'inizializzazione verrà eseguita nel momento che si importa il package
- ⇒ Se importo un modulo in un package, verrà eseguita prima l'inizializzazione del package e poi del modulo

Quando importo i moduli dei package — *from <package> import ** — non vengono importati tutti i moduli del package per motivi di performance (vorrebbe dire inizializzare tutto). Di per sé non viene importato nulla.

⇒ Si specificano i moduli di un package da importare in `__init__.py` tramite la lista speciale `__all__`
=> `__all__ = ["modulo1", "modulo2"]`

Librerie

L'insieme di package compone il nostro programma che, se è pensato per il riutilizzo in altri progetti, si chiama **libreria**. È un pezzo di codice riutilizzabile che possiamo usare importandolo nel nostro programma e semplicemente usarlo importando quella libreria e chiamando il metodo di quella libreria con un punto (tipicamente organizzata in package).

- ⇒ Oltre alle librerie standard (già incluse nell'interprete), Python ha molte librerie pronte all'uso
<https://docs.python.org/3/library/> | <https://pypi.org/>

Librerie standard

sys (già usata), *string* (manipolazione stringhe), *datetime* (manipolazione date), *math* (funzioni matematiche), ...

Su [Pypi](#) ci sono moltissime librerie – per installarle, si usa *pip3* (o *pip*) → ***pip3 install <libreria>***

pip è il packet manager (come *apt*) che installa/rimuove librerie Python.

Di default, va a cercare tali librerie su [Pypi](#) ma è possibile specificare altri repository

- ⇒ <http://www.python.it/about/applicazioni/> → librerie molto usate

L'interprete va a pescare le librerie aggiuntive per importarle dalla lista delle directory in cui cercare i package (in ordine).

➤ ***import sys sys.path***

- ⇒ La prima voce è la directory corrente, ovvero prima si guardano i package del progetto.

- ⇒ Di solito è seguita dai path presenti nella variabile d'ambiente ***PYTHONPATH*** e poi da
/usr/lib/python.x.x

Ambienti

Modificando questi path si possono caricare altri package a runtime (es. *sys.path*) o scegliere tra una versione di un package o un'altra. Spesso progetti diversi sullo stesso PC vogliono diverse versioni della stessa libreria.

Per evitare conflitti:

- ⇒ Si possono installare le due versioni in directory diverse e modificare il ***PYTHONPATH*** a seconda del progetto (oppure)
- ⇒ Creare un ambiente virtuale <https://docs.python.org/3/library/venv.html>
(se non è installato *venv*, installarlo con *pip3 install virtualenv*)
- Creazione ambiente → *python3 -m venv /path/to/new/virtual/environment*
 - Attivazione ambiente → */path/to/new/virtual/environment*
 - Disattivazione ambiente → ***deactivate***

Una volta attivato, *pip3* installerà le librerie solo in questo ambiente e l'interprete ci cercherà le librerie.

Ogni progetto dovrebbe avere il suo ambiente per evitare conflitti.

- Esportazione versioni delle librerie di un ambiente → *pip3 freeze > requirements.txt*
- Importazione versioni delle librerie di un ambiente → *pip3 install -r requirements.txt*

Moduli e script

Come abbiamo visto, i moduli eseguono codice quando vengono importati – stesso codice che verrebbe eseguito dall'interprete se il file del modulo venisse lanciato come script, ma a volte si vuole distinguere questo comportamento.

- ⇒ Es. *venv* si comporta diversamente se lanciato per creare un ambiente (come prima) o se importato per utilizzare le sue classi (<https://docs.python.org/3/library/venv.html#api>)

È possibile distinguere ciò utilizzando la variabile ***__name__***:

- Prende il nome del modulo quando è importato
- Prende ***__main__*** quando il file è lanciato come script

È buona norma mettere le istruzioni che si vogliono eseguire dentro un ***if __name__ == "__main__": ...***

⇒ In questo modo, lo stesso file può essere importato senza problemi da un altro file come se fosse un modulo
python3 -m venv /path/to/new/virtual/environment → indica che il modulo deve essere lanciato come script

- ⇒ In pratica, come fare *python3 venv.py* (ma non sapendo dove sia tale file meglio che l'interprete lo cerchi)

Build

Per distribuire il codice è buona norma buildarlo in un pacchetto (su Pypi sono uploadati in questa forma)

- *pip3 install build*
- *python3 -m build*

Non basta, deve esserci un file ***pyproject.toml*** che dia i metadati del progetto (es. l'autore, le dipendenze, ...)

- ⇒ Il *toml* può essere anche vuoto – si possono mettere molte info (<https://packaging.python.org/en/latest/guides/writing-pyproject-toml/#writing-pyproject-toml>)

Alla fine del processo, nella cartella *dist* vi sarà il pacchetto da poter distribuire.

Gestione della memoria

I dati di un programma vengono salvati in memoria (**stack** o **heap**, a seconda del punto del programma).

| | | |
|---|--|---|
| Stack → variabili dichiarate nel codice <pre>int main() { int a = 10; }</pre> | Heap → memoria esplicitamente allocata <pre>int main() { int *a = (int*)malloc(sizeof(int)); }</pre> | NB: sono esempi in C, servono solo a far capire il concetto di stack e heap. |
|---|--|---|

Problema: fino a quando la variabile deve esistere in memoria? (fino a quando deve occuparla?)

Approcci:

- **Manuale** → lo decide il programmatore
- **Automatica** → lo decide il compilatore/interprete
 - Reference counting
 - Garbage collector (generational, tracing)

Manuale

Il programmatore è responsabile delle allocazioni e deallocazioni della memoria (es. in C si usa *malloc()* e *free()*)

- ⇒ **Vantaggi:** gestione esplicita, molto veloce
- ⇒ **Svantaggi:** frequenti errori, memory leaks, puntatori a zone non allocate, segmentation faults

Reference Counting

Algoritmo che libera memoria automaticamente quando un oggetto non è più referenziato (es. in Perl e PHP).

1. Ogni oggetto ha un campo aggiuntivo che indica il numero di riferimenti ad esso
 2. Ogni volta che si aggiunge/rimuove un riferimento, il contatore viene incrementato/decrementato
 3. Quando il contatore è a 0, l'oggetto viene distrutto
- ⇒ **Vantaggi:** molto veloce, reattivo (non appena si raggiunge lo zero, l'oggetto viene eliminato)
 - ⇒ **Svantaggi:** mantenere aggiornato il contatore implica un overhead, problema dei riferimenti circolari

Problema dei riferimenti circolari

Due oggetti possono referenziarsi a vicenda. Il contatore di ogni oggetto è almeno 1.

Anche quando i due oggetti non sono più referenziati altrove, essi avranno comunque 1 (**non verranno mai distrutti**).

Possibile soluzione: marcare come deboli i riferimenti ricorsivi. Considerarli come tali per la rimozione:

- Li uso solo se almeno un oggetto è referenziato altrove
- Maggior overhead

Garbage Collector

Il garbage collector è un'entità che ciclicamente controlla lo stato degli oggetti in memoria e eventualmente li elimina (liberando memoria). Tra gli algoritmi utilizzati dai garbage collectors ci sono il **tracing** e il **generational**.

Tracing

Algoritmo:

- ⇒ Partendo da degli oggetti radice segue tutti gli oggetti referenziati
- ⇒ Procede a cascata
- ⇒ Gli oggetti non referenziati vengono considerati garbage ed eliminati

In questo modo vengono mantenuti in memoria tutti gli oggetti raggiungibili (direttamente o tramite altri oggetti) dagli oggetti radice. Gli oggetti radici sono: variabili globali, variabili locali e argomenti di funzione.

Un oggetto può essere non più raggiungibile per due motivi:

1. Ragione **sintattica** → la sintassi implica che quell'oggetto non sia più raggiungibile

- Es. $a = 1$; $a = 2 \Rightarrow$ l'oggetto 1 non è più raggiungibile
- = > Facile da trovare \rightarrow i garbage collectors si basano soprattutto su questa
- \Rightarrow Ragione **semantica** \rightarrow oggetti non più raggiungibili per via del flusso di codice (es. branch di codice mai utilizzati)
- = > Difficile da trovare (euristiche)

Questo tipo di garbage collectors implicano un overhead per trovare gli oggetti non più raggiungibili \rightarrow bisogna stabilire quando chiamarlo – es. ogni tot di tempo (Java) o quando la memoria occupata raggiunge un limite (Python).

Mark and Sweep

Ogni oggetto ha associato un flag: 1 (raggiungibile) o 0 (non raggiungibile)

- \Rightarrow Fase di **scansione** \rightarrow oggetti marchiati come 0 e 1
- \Rightarrow Fase di **deallocazione** \rightarrow oggetti 0 rilasciati

Limiti:

- Ogni volta si riscansiona tutto
- Ogni volta si ricicla sugli oggetti 0
- Ogni volta il programma interrompe per eseguire il Garbage Collector

Tri-color Marking

Algoritmo organizzato in tre insiemi:

- **White** \rightarrow candidati alla rimozione
- **Grey** \rightarrow oggetti raggiungibili ma i cui oggetti referenziati da essi non sono stati ancora analizzati
- **Black** \rightarrow oggetti raggiungibili che non referenziano nessun oggetto nel White Set

Gli oggetti possono avere solo il seguente flusso: White Set \rightarrow Grey Set \rightarrow Black Set

Fase 1 – Inizializzazione degli insiemi

Tutti gli oggetti radice messi nel Grey Set, mentre tutti gli altri nel White Set. Black Set vuoto.

Fase 2 – Fino a svuotare il Grey Set

Si sceglie un oggetto O dal Grey Set.

Si identificano tutti gli oggetti referenziati da O ($R1, R2, \dots$)

Tra questi si identificano quelli appartenenti al White Set ($W1, W2, \dots$)

Essi sono referenziati (da O) quindi vengono messi nel Grey set.

Ora O può essere messo nel Black Set (sono stati analizzati gli oggetti referenziati e essi non sono più nel White Set).

Fase 3 – Liberazione della memoria

Viene liberata la memoria degli oggetti nel White Set che sono:

- Non raggiungibili direttamente (fase 1) | Non raggiungibili indirettamente (fase 2)

Vantaggi: fase 1 e 2 possono essere eseguite mentre il programma gira (senza interromperlo).

Rilascio Memoria

Ci sono due strategie per liberare la memoria:

1. In movimento \rightarrow copio tutti gli elementi raggiungibili in una nuova area di memoria
2. Non in movimento \rightarrow rilascio le zone di memoria degli oggetti non raggiungibili

La strategia in movimento sembra meno efficiente, ma:

- Meno frammentazione
- Possibilità di avere più oggetti che si referenziano nella cache

Generational

Ipotesi: gli oggetti creati più di recente hanno la maggior probabilità di diventare irraggiungibili nell'immediato futuro.

I garbage collectors generazionali suddividono gli oggetti in insiemi di "vecchiaia":

- \Rightarrow Nei vari cicli di esecuzione, fa controlli frequenti solo sugli oggetti delle generazioni più giovani
- \Rightarrow Contemporaneamente, tiene traccia della creazione di riferimenti tra generazioni

È **più veloce**, ma **meno preciso** (alcuni oggetti non raggiungibili potrebbero rimanere).

Generazioni:

- **Eden** \rightarrow oggetti appena creati

- **Survivor 2** → oggetti sopravvissuti ad un certo numero di cicli
- **Survivor 1** → oggetti sopravvissuti ad un certo numero di cicli (maggiore di S2)
- **Old** → oggetti più vecchi

Gli oggetti permanenti usati dall'interprete non rientrano in questi termini.

Ogni volta che un gruppo supera una soglia viene invocato il Garbage Collector solo su quel gruppo.

Gli oggetti raggiungibili vengono copiati nell'insieme immediatamente più vecchio – la regione viene svuotata.

⇒ **Vantaggi** → agisce su set ridotti di oggetti

⇒ **Svantaggi** → minor precisione

Approcci ibridi

- Minor cycle → fatto frequentemente (es. Generational) | molto performante, minor precisione
- Major cycle → fatto ogni tanto (es. Mark and Sweep) | su tutti gli oggetti

Gestione memoria in Python

Dipende dalla versione di Python (e dall'interprete), ma ora è un approccio ibrido:

⇒ **Reference counting**

⇒ **Generational** Garbage Collector

API

Il modulo `gc` consente di impostare il Garbage Collector (<https://docs.python.org/3/library/gc.html#module-gc>)

- `gc.enable()/disable()` → abilita/disabilita il Garbage Collector
- `gc.collect(generation = 2)` → esegue il Garbage Collector su una generazione

In realtà, dipende dall'interprete – ce ne sono diversi in Python (CPython, Jython, PyPy, ...).

CPython è il più usato – scritto in C

⇒ `gc` generazionale a soglia

⇒ **Reference Counting** come principale meccanismo di gestione memoria.

| | |
|--|---|
| <pre>import sys a = 'aaa' sys.getrefcount(a) -> 2</pre> | Abbiamo un ulteriore riferimento dato dal passaggio di parametro alla funzione |
| <pre>a = 1 sys.getrefcount(a) -> 2 x = [a] d = {'a':a} sys.getrefcount(a) -> 4</pre> | Quando si esce dalla prima funzione, il contatore viene decrementato (il riferimento dell'argomento non esiste più) |

Il SO mette a disposizione diversi strumenti per **profilare il consumo di memoria**, ma non sono granulari.

Python ha diverse librerie per farlo: **memory-profiler** (`apt install python3-memory-profiler`) |

tracemalloc

```
import tracemalloc

def foo():
    f = [ x for x in range(0, 100000) ]

tracemalloc.start()

foo()

current, peak = tracemalloc.get_traced_memory()

tracemalloc.stop()

print( "Istananea ", current, " Picco ", peak )
```

La gestione della memoria a **basso livello** dipende dall'interprete e in che linguaggio è scritto.

In **CPython** esiste un allocatore di oggetti che si occupa di allocare la memoria per essi:

- Quando creo un oggetto, l'interprete chiama questo allocatore
- L'allocatore si occupa di gestire lo spazio ed effettivamente allocare la memoria tramite l'SO

Siccome in Python tutto è un oggetto, ogni volta bisognerebbe scomodare il SO per allocare piccole zone di memoria. L'allocatore Python fa da filtro e alloca zone grandi meno volte e poi le gestisce lui => gestisce un suo heap privato.

- ⇒ Quando si crea un oggetto, l'interprete lo alloca nell'heap privato (usando una funzione di allocazione propria)
- ⇒ L'allocatore si occupa di crearlo in questa zona e gli assegna una zona di memoria
- ⇒ Se serve più memoria viene chiamata una *malloc()* (viceversa, quando si rimuove un oggetto)

Se un oggetto è molto grande (> 512 byte) viene chiamata direttamente la *malloc()*.

Se l'oggetto ha un allocatore specifico viene chiamato esso.

Le zone memoria gestite dall'allocatore sono divise in **Arene** a loro volta divise in **Pool** (gestione più efficiente).

- ⇒ Quando si alloca un oggetto viene messo nell'**Arena** con più Pool pieni.
In questo modo, è più probabile che le Arene si svuotino potendole liberare dalla memoria (*free()*)
- Un Pool ha la dimensione di una pagina stabilita dal SO (memoria contigua, allocazione determinata dal SO). Al suo interno ha dei blocchi di una dimensione prefissata ed uguale.
- ⇒ Quando si alloca un oggetto si cerca (nell'Arena) il **Pool** con i blocchi della dimensione richiesta (se non esiste, viene creato)
- ⇒ Quando si libera un oggetto, il blocco viene segnato come **libero** (ogni blocco contiene un solo oggetto).

Debug

Anche in Python ci sono strumenti per effettuare il debug del codice (integrati nell'IDE oppure a se stanti).

A differenza di altri linguaggi, in Python c'è un modulo per debuggare → *pdb*

- ⇒ Eseguito nel codice

```
import pdb  
  
pdb.run(<stringa di codice>)
```

run(statement[, globals[, locals]])

Esegue l'istruzione statement (passata come stringa) sotto il controllo del debugger. Il prompt del debugger compare prima che qualunque codice venga eseguito; potete impostare dei breakpoint e digitare "continue", oppure potete avanzare un passo alla volta tra le dichiarazioni utilizzando "step" o "next" (tutti questi comandi vengono spiegati più avanti). Gli argomenti facoltativi globals e locals specificano l'ambiente nel quale il codice viene eseguito; se non specificato diversamente, viene utilizzato il dizionario del modulo `__main__`. (Vedete la spiegazione dell'istruzione `exec` o della funzione built-in `eval()`.)

runeval(expression[, globals[, locals]])

Valuta l'espressione expression (fornita come stringa) sotto il controllo del debugger. Quando `runeval()` termina, questa funzione restituisce il valore dell'espressione. Altrimenti questa funzione è simile a `run()`.

runcall(function[, argument, ...])

Esegue la funzione function (una funzione o il metodo di un oggetto, non una stringa) con gli argomenti forniti. Quando `runcall()` termina, la funzione restituisce qualunque cosa venga restituito dalla funzione chiamata. Il prompt del debugger appare al momento dell'entrata nella funzione.

- ⇒ Lanciato come script → *python3 -m pdb <programma>*

La maggior parte dei comandi possono venire abbreviati con una o due lettere; per esempio "h(elp)" significa che sia "h" che "help" possono venire utilizzati per avviare l'help (ma non "he", "hel", "H", "Help" o "HELP").

Gli argomenti dei comandi devono essere separati da caratteri di spaziatura (spazi o tab).

Nella sintassi dei comandi gli argomenti facoltativi vengono racchiusi tra parentesi quadre ("[]"); le parentesi quadre non devono essere digitate. Le varie alternative nella sintassi dei comandi vengono separate da una barra verticale ("|"). Inviando una riga vuota (invio) si otterrà la ripetizione dell'ultimo comando fornito.

- Eccezione: se l'ultimo comando era "list", vengono elencate le prossime 11 righe.

Comandi

| |
|--|
| w(here) |
| Stampa la traccia dello stacke, con il frame più recente in fondo. Una freccia indica il frame corrente, che determina il contesto della maggior parte dei comandi. d(own) Sposta il frame corrente in basso di un livello nella traccia dello stack(verso un frame più recente). |
| u(p) |
| Sposta il frame corrente in alto di un livello nella traccia dello stacke(verso un frame più vecchio). |
| b(reak) [[filename:]lineno function[, condition]] |
| Con un argomento lineno, imposta in quella riga del file corrente un break. Con un argomento function, imposta un break alla prima istruzione eseguibile in quella funzione. Il numero di riga può essere preceduto da un nome di file seguito da un due punti, in modo da specificare un breakpoint in un'altro file (probabilmente uno che non è ancora stato caricato). Il file viene cercato in sys.path. Notate che ad ogni breakpoint viene assegnato un numero a cui fanno riferimento tutti gli altri comandi dei breakpoint. Se è presente un secondo argomento, esso è un'espressione che deve venire valutata come vera prima che il breakpoint venga rispettato. Senza argomenti, vengono elencati tutti i breakpoint, includendo per ognuno di essi il numero di volte che è stato raggiunto, il contatore corrente dei passi da ignorare e la condizione associata, se presente. |
| tbreak [[filename:]lineno function[, condition]] |
| Breakpoint temporaneo che viene rimosso automaticamente quando viene raggiunto la prima volta. Gli argomenti sono gli stessi di break |
| cl(ear) [bpnumber [bpnumber ...]] |
| Con una lista di numeri di breakpoint separata da spazi, cancella tutti questi breakpoint. Senza argomenti, cancella tutti i breakpoint (ma prima chiede conferma). |
| disable [bpnumber [bpnumber ...]] |
| Disabilita i breakpoint forniti come lista di numeri di breakpoint separati da spazi. Disabilitare un breakpoint significa che esso non può più provocare il blocco dell'esecuzione del programma, ma al contrario della cancellazione del breakpoint, esso resta nella lista dei breakpoint e può venire riabilitato. |
| enable [bpnumber [bpnumber ...]] |
| Abilita i breakpoint specificati. |
| ignore bpnumber [count] |
| Imposta il contatore dei passi da ignorare per il breakpoint fornito (come numero). Se count viene omissso, il contatore dei passi da ignorare viene impostato a 0. Un breakpoint diventa attivo quando il contatore dei passi da ignorare diventa 0. Quando diverso da zero, il contatore viene decrementato ogni volta che il breakpoint viene raggiunto, a condizione che esso non sia disabilitato e che ogni condizione associata sia stata valutata come vera. |
| condition bpnumber [condition] |
| condition è un'espressione che deve essere valutata come vera prima che il breakpoint venga rispettato. Se condition è assente, tutte le condizioni esistenti vengono rimosse; cioè, il breakpoint viene reso incondizionato. |
| s(tep) |
| Esegue la riga corrente, blocca l'esecuzione alla prima occasione possibile (in una funzione chiamata o sulla prossima riga della funzione corrente). |
| n(ext) |
| Continua l'esecuzione finché la prossima riga della funzione corrente non viene raggiunta o la funzione termina. (La differenza tra "next" e "step" è che "step" si blocca dentro una funzione chiamata, mentre "next" esegue la funzione chiamata a (circa) piena velocità, fermandosi solo alla prossima riga nella funzione corrente.) |
| r(eturn) |
| Continua l'esecuzione fino al termine della funzione corrente. |
| c(ontinue) |
| Continua l'esecuzione, si blocca solo quando viene raggiunto un breakpoint. |
| j(ump) lineno |
| Imposta la prossima riga che verrà eseguita. Disponibile solo nel frame più in basso. Questo permette di tornare indietro per rieseguire una porzione di codice o per saltare del codice che non volete eseguire. Attenzione che non tutti i salti sono permessi; tanto per chiarire, non è possibile saltare nel mezzo di un ciclo for o fuori da una clausola finally. |

| |
|--|
| l(ist) [first[, last]] |
| Mostra il codice sorgente del file corrente. Senza argomenti, mostra le 11 righe attorno a quella corrente o continua l'elenco precedente. Con un argomento, elenca le 11 righe attorno a quella riga. Con 2 argomenti, mostra le righe nell'intervallo fornito; se il secondo argomento è minore del primo, viene interpretato come un incremento(n.d.T: 11 e 3 indicano le righe dalla 11 alla 14). |
| a(rgs) |
| Stampa la lista degli argomenti della funzione corrente. |
| p expression |
| Valuta l'espressione expression nel contesto corrente e ne stampa il valore. Note: Si può anche utilizzare "print", ma non è un comando del debugger; esso esegue l'istruzione print di Python. |
| pp expression |
| Come il comando "p", ad eccezione del fatto che il valore dell'espressione viene stampato in forma elegante utilizzando il modulo pprint. |
| alias [name [command]] |
| <p>Crea un alias chiamato name che esegue il comando command. Il comando non deve essere racchiuso tra virgolette. I parametri sostituibili possono venire indicati da "%1", "%2" e così via, mentre "%*" viene sostituito da tutti i parametri. Se non viene fornito nessun comando, viene mostrato l'alias corrente di name. Se non viene fornito nessun argomento, vengono elencati tutti gli alias. Gli alias possono venire annidati e possono contenere qualsiasi cosa che sia possibile digitare al prompt di pdb. Notate che i comandi interni di pdb possono venire sovrascritti dagli alias. Un comando sovrascritto rimane perciò nascosto finché l'alias non viene rimosso. Il meccanismo degli alias viene applicato ricorsivamente alla prima parola della riga di comando; tutte le altre parole sulla stessa riga di comando vengono lasciate invariate. Come esempio, ecco due utili alias (specialmente quando inseriti nel file .pdbrc):</p> <pre>#Visualizza le variabili d'istanza (utilizzo: "pi classInst") alias pi for k in %1.__dict__.keys(): print "%1.",k,"=",%1.__dict__[k] #Visualizza le variabili d'istanza in self alias ps pi self</pre> |
| unalias name |
| Cancella l'alias specificato. |
| [!]statement |
| <p>Esegue l'istruzione (monoriga) statement nel contesto dello stack frame corrente. Il punto esclamativo può venire omesso, a meno che la prima parola dell'istruzione sia anche un comando del debugger. Per impostare una variabile globale potete precedere, sulla stessa riga, il comando d'assegnamento con un comando "global", per esempio:</p> <pre>(Pdb) global list_options; list_options = ['-l'] (Pdb)</pre> |
| q(uit) |
| Esce dal debugger. Il programma in esecuzione viene interrotto. |

È possibile definire porzioni di codice che vengano eseguite in debug

```
if __debug__:
```

```
...
```

La `__debug__` è definita sempre tranne quando si utilizza l'opzione `-O` che indica all'interprete di ottimizzare il codice per la produzione. Modo equivalente ma più rapido → ***assert***

```
assert <expr> -> if __debug__:
```

```
    if not expr: raise AssertionError
```

```
assert <expr1> <expr2> -> if __debug__:
```

```
    if not expr1: raise AssertionError(expr2)
```

Unit Test

Lo Unit Testing è il primo livello di test del software in cui vengono testate le parti (testabili) più piccole di un software. Questo viene utilizzato per convalidare che ogni unità del software funzioni come previsto.

Test Case (caso di prova)

Un caso di test è un insieme di condizioni che viene utilizzato per determinare se un sistema sottoposto a test funziona correttamente.

Test Suite (serie di test)

La suite di test è una raccolta di casi di test che vengono utilizzati per testare un programma software per dimostrare che ha una serie specifica di comportamenti eseguendo insieme i test aggregati.

Test Runner (esecutore di test)

Un test runner è un componente che imposta l'esecuzione dei test e fornisce il risultato all'utente.

In Python si utilizza il modulo *unittest*

```
import unittest

value = True # False

class Foo(unittest.TestCase):
    def test(self):
        self.assertTrue(value)

unittest.main()
```

Un Test Case è una classe che eredita da *TestCase*.

Al suo interno vi sono i test da effettuare (che iniziano con la parola *test*).

Per effettuare la valutazione si usano le *assert* messe a disposizione dal modulo.

- Se esse non vanno a buon fine il test viene considerato NON superato.

Ci sono 3 tipi di possibili risultati del test:

1. **OK** → tutti i test sono stati superati
2. **FAIL** → il test non è passato e viene sollevata un'eccezione *AssertionError*
3. **ERRORE** → il test solleva un'eccezione diversa da *AssertionError*

La classe può avere anche metodi per preparare i test da eseguire operazioni dopo l'esecuzione

- Es. connessione ad un database, chiusura di un file, ...
- ⇒ *setUp()* → eseguito prima di ogni test
- ⇒ *tearDown()* → eseguito dopo ogni test
- ⇒ *setUpClass()* → eseguito prima di tutti i test della classe
- ⇒ *tearDownClass()* → eseguito dopo tutti i test della classe

File I/O

Sulle unità di memorizzazione (dischi/chiavette/...) i dati sono registrati in blocchi contenenti sequenze di lunghezza fissa composte da byte.

È una componente del SO che fornisce una "visione" comoda per accedere ai dati, ovvero una visione a file, contenuti nelle directory.

- ⇒ Ogni file ha un nome (o più precisamente, almeno un nome)
- ⇒ Per individuare un file nel file system occorre un pathname (nome del percorso)
- ⇒ A partire dalla radice si possono nominare tutte le directory da attraversare per "raggiungere il file"
- Per evitare di specificare sempre il percorso a partire dalla radice ogni processo ha una directory di lavoro.
- Di norma, un processo eredita la stessa directory di lavoro del processo che l'ha attivato.

Per aprire un file si usa l'istruzione "*open*", per chiuderlo si usa il metodo "*close*".

open

Ha due parametri:

1. Il pathname del file
2. Il modo di apertura → '*r*' (lettura), '*w*' (scrittura, azzerando il contenuto), '*a*' (aggiunta)
→ '*b*' (per aprire il file in modalità binaria) — da **aggiungere** ad uno dei precedenti

Esempio - Lettura

Si apre il file così, il valore di uscita della open è il descrittore del file da usare per individuare il file aperto.

```
– fd=open("miofile","r")
```

Si può leggere l'intero file in una stringa:

```
– filecontents=fd.read()
```

Si può leggere una riga (e poi le successive con identico metodo)

```
– fileline=fd.readline()
```

Si può fare un ciclo usando il file come sequenza, a ogni iterazione "line" sarà istanziato ad una riga del file

```
– for line in fd:
```

Alla fine il file si chiude così:

```
– fd.close()
```

Esempio – Scrittura

Si apre il file così, il valore di uscita della open è il descrittore del file da usare per individuare il file aperto.

```
– fd=open("miofile","w")
```

Si può scrivere una stringa:

```
– fd.write(s)
```

Si può usare il parametro file= della print (solo in Python3)

```
– print("hello world", file=fd)
```

Alla fine il file si chiude così:

```
– fd.close()
```

Esempio – Modalità binaria

In modalità binaria si salvano bytes grezzi. È compito del programmatore saperli poi interpretare:

```
prova = open('prova.bin', 'wb')
```

```
prova.write(bytes([1, 2, 3]))
```

```
prova.close()
```

```
prova2 = open('prova.bin', 'rb')
```

```
a = prova2.read()
```

```
a = list(a)
```

L'istruzione ***with*** consente una sintassi alternativa

```
with open("file","r") as fd:
```

....

Nel blocco dipendente dall'istruzione ***with***, *fd* è il descrittore del file aperto.

Se avviene un errore il blocco non viene eseguito e comunque il file viene chiuso alla fine del blocco.

Pickle

Modulo che permette di salvare oggetti arbitrari su file, serializzando il dato (è comune nel trasmettere/salvare dati).

⇒ `pickle.dump(obj, file)` → salva l'oggetto su file

⇒ `pickle.dumps(obj)` → restituisce la serializzazione come stringa

⇒ `pickle.load(file)` → legge il file e restituisce l'oggetto

⇒ `pickle.loads(data)` → legge la stringa *data* e restituisce l'oggetto

Pickle può essere pericoloso: potremmo deserializzare un oggetto che contiene codice malevolo. Farlo con oggetti fidati.

Si possono salvare: built-in; integers, floating-point numbers, complex numbers; strings, byte, bytearrays; tuples, lists, sets e dictionaries contenenti oggetti salvabili; funzioni (no *lambda*); classi; funzioni che abbiano il metodo

`__getstate__` salvabile (di default ritorna il `__dict__`) (utile per salvare solo alcuni campi)

NB: le funzioni e classi vengono salvate utilizzando solo il loro nome (non il codice per motivi di sicurezza). Questo implica che quando vengono deserializzate è necessario importarle preventivamente.

Performance: C e Python

In Python è possibile profilare i tempi di esecuzione con `python3 -m cProfile foo.py`

È possibile anche profilare specifiche parti di codice

```
import cProfile, pstats, io

from pstats import SortKey

pr = cProfile.Profile()

pr.enable()

# ... do something ...

pr.disable()

s = io.StringIO()

sortby = SortKey.CUMULATIVE

ps = pstats.Stats(pr, stream=s).sort_stats(sortby)

ps.print_stats()

print(s.getvalue())
```

Anche in C è possibile misurare i tempi di esecuzione

```
#include <sys/time.h>

int main() {

    long start, end;

    struct timeval timecheck;

    gettimeofday(&timecheck, NULL);

    start = (long)timecheck.tv_sec * 1000 + (long)timecheck.tv_usec / 1000;

    usleep(200000); // 200ms

    gettimeofday(&timecheck, NULL);

    end = (long)timecheck.tv_sec * 1000 + (long)timecheck.tv_usec / 1000;

    return 0;

}
```

Spesso, quindi, si scrive un programma in Python e poi si utilizzano funzioni C per le parti più onerose:

- Codice semplice per la maggior parte del programma
- Parte più lenta diventa molto più veloce

Sappiamo già che le performance non sono compatibili: C è molto più veloce, Python è molto più comodo.

Si crea una funzione C e la si compila con `cc -fPIC -shared -o my_functions.so my_functions.c`

```
int square(int i) {
    return i * i;
}
```

Si scrive il programma Python che carica la libreria C e la utilizza

```
from ctypes import *

so_file = "/Users/pankaj/my_functions.so"

my_functions = CDLL(so_file)

print(my_functions.square(10))

print(my_functions.square(8))
```

Parsing argomenti

È comune dover passare degli argomenti in input ad un programma.

- ⇒ `./la.out primo_arg secondo_arg`
- ⇒ `python3 ./script.py primo_arg secondo_arg`

In Python è possibile ricavare gli argomenti utilizzando il modulo `sys` – una volta importato, gli argomenti si trovano in `sys.argv` sottoforma di lista. Usarlo implica parsare a mano tutti gli argomenti.

- ⇒ Tenere conto della posizione e del tipo di dato in ingresso
- ⇒ Controllare il tipo di dato oppure parsare eventuali = per l'assegnazione di un parametro

Il modulo `argparse` gestisce automaticamente questi aspetti.

Vi è una fase in cui si definisce cosa ci si aspetta come argomenti. Da questa definizione il modulo:

- Genera un help
- Parsa gli argomenti nel modo corretto

Fase di definizione

Si crea un oggetto *ArgumentParser* con una descrizione generica del programma

```
import argparse
```

```
parser = argparse.ArgumentParser(description='Process some integers.')
```

Con *add_argument* si aggiungono gli argomenti che ci si aspetta in input

```
parser.add_argument(dest='integers', metavar='N', type=int,
```

```
nargs='*', help='an integer for the accumulator')
```

Fase di parsing

Con *parse_args* si analizzano i parametri passati al programma da linea di comando.

⇒ Ritorna un oggetto Namespace con le variabili popolate

```
arguments = parser.parse_args()
```

add_argument()

Versione base → *parser.add_argument('name')*

⇒ Il primo parametro finirà in *name*

⇒ Serve per gestire i parametri posizionali (considerati obbligatori)

parser.add_argument('f', '--foo')

⇒ Specifica il nome dell'argomento nella versione corta e lunga (considerati opzionali)

Parametri opzionali:

- **dest** → specifica la variabile di destinazione (di default usa il nome del parametro)
- **type** → tipo di dato (default stringa) che ci si aspetta in quel parametro = > *argparse.FileType('r')* per i file
- **default** → valore di default per i parametri opzionali
- **nargs** → numero di valori da considerare per quell'argomento
 - *nargs = 2* → *-foo a b* → *foo = ['a', 'b']*
 - *** per tutti i valori successivi (fino al nuovo argomento definito)
- **action** → azione da compiere sui valori dell'argomento
 - **store** → default, salva il valore
 - **store_true/false** → salva True o False a seconda se il parametro è stato specificato (o viceversa). Utilizzato per controllare argomenti flag.
 - **save_const** → salva il valore definito in '*const*'. Usato per controllare argomenti flag ma con un valore diverso da True se l'argomento è presente.
 - **append** → crea una lista se il parametro è specificato più volte
 - **count** → conta il numero di volte che il parametro è specificato (es. *-v v v* → *v = 3*)
- **help** → specifica una stringa da mostrare nell'help per quell'argomento
- **metavar** → valore di esempio da mostrare nell'help per quell'argomento