IL LINGUAGGIO PYTHON

0. Introduzione

Questo documento propone nozioni fondamentali per imparare a conoscere il linguaggio di programmazione Python. Attualmente la crescita riguardante l'utilizzo di questo linguaggio è notevole, in quanto ha superato il tanto utilizzato Java. Spesso Python lo si mette in competizione a Java in quanto sono due linguaggi di programmazione ad oggetti.
È importante conoscere Python nel 2020 in quanto è molto utilizzato dalle grandi aziende in ambito Web, e inoltre è il più utilizzato e più famoso in ambito di "Intelligenza Artificiale".
Il documento viene fornito in modalità gratuita e non è possibile effettuare qualsiasi attività economica di lucro su di esso.

Questo testo è scritto da Daniel Rossi, studente di ingegneria informatica presso il dipartimento di ingegneria dell'università di Modena e Reggio emilia.

Contatti:

Instagram: @DanielrossiTV / @OfficialProjecto

YouTube: Https://youtube.com/c/ProjectoOfficial?sub_confirmation=1

Al link del canale YouTube potete trovare videocorsi e progetti riguardanti il linguaggio di programmazione Python.

1. Caratteristiche e differenze dagli altri linguaggi

Innanzi tutto Python è un linguaggio dinamico a differenza ad esempio del C che è un linguaggio statico:

- *Linguaggio statico:* è un linguaggio ad alto livello in cui le operazioni effettuate a run-time sono legate quasi esclusivamente all'esecuzione del codice del programmatore (un'eccezione notevole è la gestione dello stack). Alcuni esempi possono essere il linguaggio C e il linguaggio Assembly;
- *Linguaggio dinamico:* è un linguaggio di alto livello in cui le operazioni effettuate a run-time non sono legate esclusivamente all'esecuzione del codice. Alcuni

linguaggi dinamici sono il Perl, Python (come stiamo dicendo), Ruby, PHP, Javascript.

In realtà non esiste una definizione univoca che differenzia un linguaggio statico da uno dinamico, nonostante ciò si possono elencare le più importanti differenze:

- Tipizzazione dei dati (attribuire ad una variabile un tipo, es float, int… viene svolta a run-time)
- Metaprogramming (svolta a run-time)
- Gestione dinamica della memoria e degli errori (svolta a run-time)
- Modello di generazione del codice (prodotto intermedio es. bytecode)

Alcune di queste caratteristiche si possono ritrovare in altri linguaggi non propriamente considerati dinamici, ad esempio la gestione della memoria in Java.

Caratteristiche di un linguaggio statico

Esiste una fase detta *"di compilazione"*, in cui il codice sorgente viene tradotto in un formato esclusivo per l'architettura considerata.
La traduzione da codice sorgente a codice macchina è 1:1, questo significa che è una rappresentazione molto fedele.
Permette l'esecuzione ad una velocità elevata.
I tipi di dati sono identificati a tempo di compilazione (non quando il programma è in esecuzione come accade in python) e non sono mutabili a tempo di esecuzione.
Non sempre fornisce strumenti di controllo, né semplificazioni; quasi tutto è lasciato al programmatore (memoria, tipizzazione). Basti pensare alla funzione malloc del C per allocare memoria a un puntatore.

Caratteristiche di un linguaggio dinamico

Nella fase di compilazione il codice sorgente viene tradotto in un formato intermedio, indipendente dall'architettura (es. bytecode)

[Bytecode: è più astratto che si pone in mezzo tra il nostro linguaggio di programmazione e il vero linguaggio macchina, e viene usato per descrivere le operazioni che costituiscono un programma. Viene chiamato così perché spesso le operazioni hanno un codice che occupa un solo Byte, anche se la lunghezza dell'istruzione può variare perché ogni operazione ha un numero specifico di parametri sui quali operare. Questo linguaggio viene utilizzato per creare indipendenza dall'hardware e facilita la creazione di interpreti]

Il formato intermedio è interpretato. Questo significa che un linguaggio dinamico è un linguaggio portatile, che funziona su qualsiasi macchina.

L’interprete sfrutta funzioni interne per gestire la memoria e gli errori in modo automatico a run-time.
Ha una tipizzazione dinamica dei dati: il tipo di una variabile (ma anche quello di una funzione) può cambiare a run-time.

Inoltre include meccanismi che permettono al programma di “analizzarsi” e modificarsi durante l’esecuzione (metaprogramming). Il Meta-Programming dà la possibilità di costruire funzioni e classi il cui obiettivo principale è la manipolazione del codice:

- Permette di *generare* nuovo codice
- Effettua *modifiche* e *wrapping* di codice esistente

[Wrapping: alcune funzioni svolte dal wrapper sono: allocare e deallocare risorse, controllare le pre-condizioni e post-condizioni, memorizzazione nella cache / riciclaggio a seguito di un calcolo lento]

Alcuni meccanismi fondamentali per il meta-programming sono:

- Decoratori
- Metaclassi
- Decoratori di Metaclassi

Altre caratteristiche

Dà la possibilità di creare strutture di dati, anche eterogenee, variabili nel tempo.
Inoltre esistono tante librerie esterne e facilmente utilizzabili per diversi compiti:

- Calcolo scientifico
- Interfacce grafiche complesse
- Supporto per il Web

Linguaggio dinamico VS statico

Come *VANTAGGI* abbiamo in primis che è più semplice da imparare per anche i neofiti. La scrittura del codice, di un software, diventa la scrittura effettiva del suo scheletro. Esistono tante librerie esterne e quindi si ha un forte riutilizzo del codice. La prototipazione, ovvero creare modelli di sviluppo del software, è veloce. Infine è *flessibile* e *portatile*.

Come *SVANTAGGI* possiamo notare la lentezza (a causa delle molte operazioni a run-time). Per via della semplicità di scrivere codice, può portare a scrivere pessimo codice (soprattutto per i principianti); è da sottolineare che python ha proprie metodologie di programmazione.

2. Tipizzazione dei dati

Come già sappiamo, ogni dato/variabile ha un tipo (ad es. float, int, double …). In un programma, ad ogni entità è associata un informazione su:

- *Valore*: valore (attuale, calcolato, …)
- *Tipo*: indicatore del tipo di dato associato
- *Dimensione*: l'area di memoria occupata

In Python tutte le variabili sono oggetti, esiste la classe Int, la classe Float, ecc… .
Il tipo di dato ha conseguenze sull'insieme di valori che un entità può assumere (valori ammissibili), la sua dimensione in memoria e e le operazioni che su tali valori si possono effettuare.

Dall'inglese: type checking, la tipizzazione è un processo che cerca di 'capire' qual è il tipo di un dato, sia esso dichiarato o prodotto/calcolato, in modo da poter effettuare i relativi controlli, e quindi

- definire i vincoli sui valori ammissibili (non puoi associare 3,14 a una variabile Int)
- definire le operazioni consentite (non puoi sommare 3,14 a una variabile Int)

Queste operazioni garantiscono che un programma sia type-safe.

Tutti i linguaggi di programmazione di alto livello hanno un proprio sistema per la tipizzazione dei dati (ma con significative differenze).

Che vantaggi porta la tipizzazione?

Ha conseguenze sulla *SICUREZZA*, quindi l'uso della tipizzazione permette di trovare codice privo di senso o illecito.
A livello di compilazione riduce i rischi di errori o risultati inattesi a run-time, ad esempio consideriamo:

variabile = 3 / "Hello World"

il primo operando è un intero, il secondo è l'operatore di divisione, e il terzo è una stringa. È un'operazione che non ha senso e quindi viene riportato un *"type error"*.

Ha conseguenze sull'*OTTIMIZZAZIONE*, quindi in uno stadio iniziale (a livello di compilazione), le informazioni di type checking possono fornire informazioni utili per applicare tecniche di ottimizzazione del codice.

Es. istruzione x * 2 può essere ottimizzata nel seguente modo per produrre uno shift più efficiente:

mul x,2 → shift_left x

Questo è possibile solo se si sa con certezza che x rappresenta un valore di tipo intero.

Ha conseguenze sull'*ASTRAZIONE* dove il meccanismo dei tipi di dato permette di riprodurre programmi ad un livello di astrazione più alto di quello di linguaggi a basso livello. Un caso limite di assenza di tipizzazione è un codice macchina nativo, quindi sequenze di bit o byte che rappresentano un unico tipo.

Ad esempio una stringa di caratteri è vista a basso livello come una sequenza di byte. In Python una stringa è un oggetto che ha metodi e attributi.

Ha conseguenze sulla *DOCUMENTAZIONE* e quindi nei sistemi di tipizzazione più espressivi, i nomi dei tipi di dato possono servire come fonte di documentazione del codice. Il tipo di dato mostra la natura di una variabile, ed in ultima analisi l'intento del programmatore.

[siccome in Pyhon esistono le "variabili senza tipo" può essere un problema utilizzarle in quanto chi va a rileggere il codice può non capire cosa contengono e di che tipo sono]

Ad esempio i Booleani, Timestamp o marcatore temporale (solitamente interi a 21/64 bit)
Dal punto di vista della comprensione del codice è diverso leggere:
int a = 32; o timestamp a = 32

è inoltre possibile la definizione di nuovi tipi di dato (typedef in C) per aumentare l'espressività del linguaggio.

Ha conseguenza sulla *MODULARITÀ* sull'uso appropriato dei tipi di dato che costituisce uno strumento semplice e robusto per definire interfacce di programmazione (API = Application Program Interface). Le funzionalità di una data API sono descritte dalle signature (ovvero i prototipi) delle funzioni pubbliche che la compongono. Leggendo le signature, il programmatore si fa immediatamente un'idea di cosa può o non può fare!
Boolean somma(int a, int b)
somma(a,b)

Quando avviene il Type Checking?

Le operazioni di type checking che stabiliscono il tipo del dato associato a porzioni di codice e ne verificano i vincoli (range di valori ammissibili e operazioni consentite) possono avvenire:

1. a tempo di compilazione (compile time)
2. a tempo di esecuzione (run-time)

Classificazione non esclusiva: sono possibili anche metodi intermedi.

Type Checking statico

Il meccanismo di type checking è *statico* se le operazioni di type checking sono eseguite **solo** a tempo di compilazione!

I linguaggi che effettuano type checking statico sono:

C , C++ , Fortran , Pascal , GO

Il type checking statico permette:

- di individuare parecchi errori con largo anticipo (inteso come una forma più sicura di verifica dell'integrità un programma)
- migliori prestazioni (ottimizzazione e mancanza di controlli a run-time)

Type Checking dinamico

Il type checking è dinamico se la maggior parte delle operazioni (non per forza tutte) di type checking vengono effettuate a tempo di esecuzione. Questo implica che non è necessario esplicitare il tipo di una variabile, ci penserà l'interprete a tempo di esecuzione ad associare alla variabile il tipo corretto in base al dato che le stiamo assegnando.

Linguaggi a type checking dinamico:
Javascript , Perl , PHP , Python , Ruby

Questa dinamicità permette più flessibilità rispetto al modello statico, con maggior overhead (più lavoro da parte dell'interprete nel gestire le risorse)

- le variabili possono cambiare tipo a run-time
- gestione della struttura dei dati a run-time (problemi di gestione della memoria, processi, tempo di esecuzione …)

Tramite il meccanismo di tipizzazione a run-time, si possono effettuare assegnamenti "arditi", ad esempio: var = <token letto da input>

dove token può avere un qualunque tipo.
Il type checker dinamico assegna il tipo corretto a run time, questo ovviamente non è possibile in linguaggi come il C o Java.

Nonostante ciò, tutta questa flessibilità ha un prezzo. Un type checker dinamico dà minori garanzie a priori perché opera (la maggior parte del tempo) a run-time, quindi è molto possibile incorrere in tante situazioni di errore a causa di input inaspettati.
Sarà quindi necessario:

- gestire le situazioni di errore a runtime con un meccanismo di gestione delle eccezioni (try: / except: … import traceback as t, t.print_stack())
- verificare la correttezza di un programma effettuando tanti test

Type checking ibrido

Alcuni linguaggi fanno un uso ibrido di tecniche di type checking statico e dinamico. In Java il type checking è completamente statico sui tipi ma dinamico per le operazioni di binding tra metodi e classi (legato al polimorfismo nella identificazione di una signature valida nella gerarchia delle classi).
Nonostante ciò, Java viene ritenuto lo stesso a type checking statico.

Vantantaggi a confronto

Type checking statico:

- identifica gli errori a tempo di compilazione (più veloce perché ci sono meno controlli)
- permette la costruzione di codice che esegue più velocemente

Type checking dinamico:

- permette una prototipazione più libera e rapida
- è più flessibile perché permette l'uso di costrutti considerati illegali nei linguaggi statici (Es. var = 5)
- permette la generazione di nuovo codice a runtime (metaprogrammi – es. funzione eval())

Tipizzazione forte

Un linguaggio di programmazione si dice di tipizzazione forte se impone regole rigide e impedisce usi incoerenti dei tipi di dato specificati (ad esempio le operazioni effettuate con operandi aventi tipo di dato non corretto).

Alcuni esempi possono essere la somma di interi con caratteri o un'operazione in cui l'indice di un array supera la sua dimensione.
Un linguaggio completamente a tipizzazione forte non esiste, sarebbe inutilizzabile.

Tipizzazione debole

D'altra parte esiste la tipizzazione debole, ovvero, quando un adotta la tipizzazione debole dei dati non impedisce operazioni incongruenti (ad esempi con operanti aventi tipi di dato non corretto). La tipizzazione debole fa spesso uso di operandi di conversione (casting implicito) per rendere omogenei gli operandi.

Spesso vengono usati nel C: si considerino le operazioni 'a' / 5 e 30 + "2"
Java è fortemente tipizzato, più del C e del Pascal, mentre Perl ha una tipizzazione molto debole.
Con la tipizzazione debole il risultato di una operazione può cambiare a seconda del linguaggio scelto per programmare, ad esempio:

var x:=5; var y:="37"; y + x

quanti risultati diversi possono esserci?

- In C si basa tutto sull'aritmetica dei puntatori
- in Java/Javascript, x viene convertito a stringa (x + y = "537")
- in Visual Basic e in Perl, y viene convertito a intero (x + y = 42)
- in Python non viene accettato ed esce con errore

Tipizzazione safe

Un linguaggio di programmazione adotta una tipizzazione safe (sicura) dei dati se non permette ad una operazione di casting implicito di produrre un crash, ad esempio Python.

Tipizzazione unsafe

Un linguaggio di programmazione adotta una tipizzazione unsafe (non sicura) dei dati se consente operazioni di casting che possono produrre eventi inaspettati o crash (ad esempio il C).

DUCK TYPING: Tipizzazione e polimorfismo

Il *Polimorfismo* è la capacità di differenziare il comportamento di parti di codice in base all'entità a cui sono applicati, implica quindi il riuso del codice.

Nello schema classico (ad esempio Java) di un linguaggio ad oggetti, il polimorfismo è di solito legato al meccanismo di ereditarietà. L'ereditarietà garantisce che le classi possano avere una stessa interfaccia.
In Java le istanze di una sottoclasse possono essere utilizzate al posto delle istanze della superclasse (polimorfismo per inclusione).
L'overriding dei metodi permette che gli oggetti appartenenti alle sottoclassi rispondano diversamente agli stessi utilizzi (un metodo dichiarato nella superclasse viene implementato in modi diversi nelle sottoclassi – metodi polimorfi).

In questo caso, applicare il polimorfismo e individuare la signature valida di un metodo implica:

- individuare la classe di appartenenza dell'oggetto, e cercarvi una signature valida per il metodo
- in caso di mancata individuazione, ripetere il controllo per tutte le superclassi (operation dispatching – potenzialmente risalendo fino alla classe radice – es. Object). Questo implica risalire tutta la gerarchia dell'ereditarietà e cercare la prima "classe giusta", quella con la signature valida per il metodo.

[Signature: la signature di un metodo sono un insieme di informazioni che lo identificano. Solitamente queste informazioni sono: *nome*, *numero di parametri*, *tipi di parametri*. Es. se ho due metodi con lo stesso nome che svolgono operazioni diverse, posso distinguerli a livello macchina in base al numero di parametri]

La ricerca della classe adatta nella gerarchia delle classi può essere effettuata:

- A tempo di compilazione
- A tempo di esecuzione

In molti linguaggi ad oggetti (ad esempio C++) avviene a tempo di compilazione, in quanto il costo a tempo di esecuzione è ritenuto troppo elevato.

In Java e nei linguaggi dinamici, avviene a tempo di esecuzione. Questo implica che può essere molto costoso in termini di overhead.
Nei linguaggi dinamici però esiste una alternativa: si va a sfruttare il *Duck Typing* che permette di realizzare il concetto di polimorfismo senza dover necessariamente usare meccanismi di ereditarietà (o di implementazione di interfacce condivise). "Se istanzio un oggetto di una classe e ne invoco metodi/attributi, l'unica cosa che conta è che i metodi/attributi siano definiti per quella classe". Questo meccanismo è reso possibile grazie al Type Checking dinamico, ovvero il controllo (duck test) viene effettuato dall'interprete a runtime.

Ad esempio: Animale, Cane e Gatto

Una funzione *funz()* prende in ingresso un parametro formale (non tipizzato, quindi senza tipo) e ne invoca il metodo *CosaMangia()*. Non è necessario che le classi *Cane* e *Gatto* siano sottoclassi di *Animale* per essere passate a *funz()* , basta che abbiano un metodo chiamato *CosaMangia()*.
All'interprete interessa solo che i tipi di oggetto espongano un metodo con lo stesso nome e con lo stesso numero di parametri in ingresso.

Esempio Python

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


def is_animal(var)
        var.quack()
```

definiamo la classe *Duck* con il metodo *quack* e all'esterno della classe creiamo una funzione chiamata *is_animal(var)* alla quale possiamo passare un qualsiasi dato, ma che effettuerà la print di "quack quack" soltanto se quel dato è un istanza della classe *Duck,* la quale possiede il metodo *quack()*.

In generale il polimorfismo a livello di tipi di dato permette di:

- Differenziare il comportamento dei metodi in funzione del tipo di dato a cui sono applicati
- evitare di dover predefinire un metodo, classe o struttura di dati appositi per ogni possibile combinazione di dati

tutto questo implica sempre il riuso del codice

[da notare che il riuso del codice implica minori costi di sviluppo del software al costo, in caso di errori, di propagare un bug all'interno di tutto il sistema]

In un linguaggio tipizzato dinamicamente, tutte le espressioni sono intrinsecamente polimorfe.

In conclusione a questo capitolo, quasi tutti i linguaggi dinamici adottano un meccanismo di gestione degli attributi e dei metodi degli oggetti basato su duck typing. Questo perché permette di ottenere polimorfismo senza dover per forza incorrere nell'overhead dovuto all'ereditarietà. Ovviamente anche qui si deve considerare la

possibilità di errori a tempo di esecuzione: in questo modo non è possibile imporre che gli oggetti rispettino una interfaccia comune.