Laboratorio d'Informatica

Corso di Laurea Magistrale in Ingegneria per l'Ambiente e il Territorio

A.A. 2021/2022

Docente: Lorenzo Putzu

Lezione 04

Il linguaggio Python

Definizione di più funzioni nello stesso file

- È possibile scrivere in uno stesso *file* la definizione di più funzioni, cioè una sequenza di istruzioni def, perciò dopo aver eseguito il *file* tutte le funzioni al suo interno saranno disponibili nella *shell*, e potranno essere chiamate

- Nelle istruzioni del corpo di una funzione possono comparire chiamate di altre funzioni, sia predefinite che definite dall'utente. Se si vuole chiamare una funzione predefinita appartenente a una delle **librerie** Python sarà necessario inserire **prima** della chiamata la corrispondente istruzione from-import.

- Di norma l'istruzione from-import viene inserita all'inizio del *file* che contiene la definizione delle proprie funzioni, **non** nel corpo di una di esse.

Definizione di funzioni con chiamate di funzioni

- Per poter chiamare dall'interno di una funzione un'altra funzione definita dall'utente sono disponibili due alternative:
  - la **definizione** delle due funzioni deve trovarsi nello **stesso** *file*
  - le due funzioni possono essere definite in *file* **diversi**, ma tali *file* dovranno trovarsi in una **stessa** cartella, e nel *file* che contiene la chiamata dell'altra funzione si dovrà inserire l'istruzione from nome-file import nome-funzione, dove:
    - – nome-file è il nome del *file* che contiene la definizione dell'altra funzione (senza l'estensione .py)
    - – nome-funzione è il nome di tale funzione

Variabili locali e globali

- I parametri di una funzione e le eventuali altre variabili alle quali viene **assegnato** un valore all'interno di essa sono dette locali, cioè vengono "create" dall'interprete nel momento in cui la funzione viene eseguita (con una chiamata), e vengono "distrutte" quando l'esecuzione della funzione termina.
- In particolare, se la chiamata della funzione viene scritta nella *shell*, e prima della chiamata è stata definita nella *shell* una variabile con lo stesso nome di una delle variabili della funzione:
  - le due variabili vengono associate a celle di memoria **distinte**
  - durante l'esecuzione della funzione l'interprete potrà accedere solo alla variabile associata alla funzione, e non potrà quindi usare o modificare il valore della variabile avente lo stesso nome definita nella *shell*, che manterrà il valore originale

Variabili locali e globali

- Anche le variabili definite in funzioni diverse e aventi lo stesso nome sono variabili locali.
- Questo significa che se una funzione è chiamata da istruzioni che si trovano nel corpo di un'altra funzione, ognuna di esse potrà accedere solo alle proprie variabili, e non potrà accedere a quelle dell'altra funzione, né modificarle.
- Il fatto che le variabili definite in una funzione siano locali consente di definire nuove funzioni senza preoccuparsi dell'eventuale presenza di variabili con lo stesso nome nei programmi che le useranno.
- Analogamente, quando si scrive un programma che chiama funzioni predefinite o definite dall'utente non ci si deve preoccupare dei nomi dei parametri e delle eventuali variabili definite in tali funzioni.

Variabili locali e globali

- Se invece all'interno di una funzione il nome di una variabile (che non sia uno dei parametri) compare in un'espressione senza che in precedenza nella funzione sia stato **assegnato** a essa alcun valore, tale variabile è considerata globale, cioè l'interprete assume che il suo valore sia stato definito nelle istruzioni precedenti la **chiamata** della funzione (scritte nella *shell* o in un programma); in caso contrario si ottiene un messaggio d'errore.

- In questo modo le istruzioni di una funzione possono accede al valore di variabili definite nella *shell* o nel programma chiamante. Tuttavia è preferibile **evitare** l'uso di variabili globali nelle funzioni, poiché la loro presenza rende più difficile assicurare la correttezza di un programma e comprenderne il funzionamento.

Struttura dei programmi: update

- In un *file* è anche possibile scrivere un programma (cioè una sequenza di istruzioni) che contenga sia istruzioni def per la definizione di funzioni, che altre istruzioni che chiamano tali funzioni.
- L'unico vincolo è che ogni chiamata compaia **dopo** la definizione della funzione corrispondente. Perciò di norma le definizioni di funzioni vengono scritte all'inizio del *file*.

```
from math import pi

def circ (raggio) :
    circonferenza = 2*pi*raggio
    return circonferenza

r = input ("Inserire il raggio: ")
print "La circonferenza è", circ(r)
```

Struttura dei programmi: update

- Come si è detto in precedenza, nei linguaggi di alto livello la definizione di nuove funzioni consente di **strutturare** un programma suddividendolo in diversi "sottoprogrammi" (funzioni), ciascuno dei quali esegue un'operazione distinta e indipendente dagli altri.
- Questo stile di programmazione è detto modulare.
- In particolare, è buona norma cercare di suddividere un programma in funzioni che siano il più possibile **brevi**.
- La programmazione modulare presenta diversi vantaggi:
  - semplifica la scrittura, la comprensione e la modifica dei programmi
  - rende più facile l'individuazione di eventuali errori
  - consente di usare una **stessa** funzione in programmi **diversi**

Struttura dei programmi: update

- Un programma Python può essere strutturato suddividendolo in una o più funzioni e in una (breve) sequenza di istruzioni da eseguire all'avvio del programma.

- Le funzioni e le istruzioni del programma "principale" potranno trovarsi in uno o più *file*. Nel caso di più *file* si dovranno prevedere opportune istruzioni from-import.

- Per eseguire il programma si dovrà eseguire il *file* che contiene le istruzioni del programma "principale".

- In alternativa, anche le istruzioni del programma "principale" potranno essere scritte sotto forma di funzione. In questo caso il programma potrà essere avviato chiamando dalla *shell* tale funzione.

Esercizi

Scrivere un programma modulare che utilizzi le funzioni definite per gli esercizi della lezione precedente (Lez03_es06, Lez03_es07). Il programma dovrà chiedere all'utente di inserire un numero n e dovrà:

- verificare se si tratta di un numero intero non negativo
- restituire il valore del fattoriale per n
- restituire la somma dei primi n termini della serie armonica
- stampare a video entrambi i risultati

Testare tutte le soluzioni analizzate

1. Funzioni e programma principale in un unico file
2. Funzioni e programma principale in file separati

Tipi di dati strutturati

- I tipi di dato vengono classificati a loro volta in:
  - tipi semplici
  - tipi strutturati
- Un tipo **semplice** è composto da valori che non possono essere "scomposti" in valori più semplici ai quali sia possibile accedere attraverso operatori o funzioni del linguaggio.
  - Esempi di tipi semplici sono i numeri interi, i numeri frazionari e i valori Booleani.

Tipi di dati strutturati

- Un tipo **strutturato** è invece composto da valori che sono a loro volta collezioni o sequenze di valori più semplici.
  - Le stringhe sono un tipo strutturato: sono infatti composte da **sequenze ordinate** di caratteri a ciascuno dei quali è possibile accedere **individualmente**, come si vedrà più avanti.

- Oltre alle stringhe, due dei principali tipi strutturati del linguaggio Python sono:
  - le liste, che consentono di rappresentare **sequenze ordinate** di valori qualsiasi
  - i dizionari, che consentono di rappresentare **collezioni** (non ordinate) di valori qualsiasi

Le liste

- In molte applicazioni i dati da elaborare sono costituiti da **sequenze ordinate** di valori più semplici.
- Per esempio, le coordinate di un punto o gli elementi di un vettore in un dato sistema di riferimento possono essere rappresentati come una sequenza ordinata di numeri reali.
- Nei programmi è conveniente poter rappresentare dati di questa natura come singoli valori **composti**, per esempio associando le coordinate di un punto in uno spazio a tre dimensioni a una **singola** variabile invece che a tre variabili distinte.
- Come si è già visto, nel caso particolare delle sequenze di caratteri questo è reso possibile dal tipo di dato *stringa*.

Le liste

- Il tipo *lista* fornisce questa possibilità per sequenze ordinate di valori che possono appartenente a tipi **qualsiasi**, anche diversi tra loro.
- Una lista si rappresenta nei programmi Python come una sequenza ordinata di valori scritti tra parentesi **quadre** e separati da virgole
- Esempi:
  - [7, -2, 4]

  una lista composta da tre numeri interi
  - [-5.3, 6, True]

  una lista composta da un numero reale, un numero intero e un valore logico
  - []

  una lista vuota

Le liste

- Una lista può essere assegnata direttamente ad una variabile come una qualsiasi espressione
  - vettore = [7, -2, 4]
- E può anche essere acquisita attraverso la tastiera
  - v = eval(input("Inserire una lista: "))
- Poiché gli elementi di una lista possono essere valori di un tipo qualsiasi, possono a loro volta essere delle altre espressioni o delle altre liste: inquest'ultimo caso si parla di liste **nidificate**. Per esempio dopo l'esecuzione delle seguente sequenza di istruzioni:
  - x = 2
  - y = -5
  - z = [y**2 + 1, "ciao",x == 3, ["a", "b"]]
  - la variabile z sarà associata ad una lista di **quattro** elementi: un numero intero, una stringa, un valore logico ed una lista di due elementi: [26, "ciao", False, ["a", "b"]]

Le liste: indicizzazione

- Come per gli altri tipi di dato, anche per le liste il linguaggio Python mette a disposizione dei programmatori diversi operatori e funzioni predefinite.
- In particolare, essendo le liste un tipo strutturato, alcuni operatori consentono l'accesso ai **singoli** elementi di una lista. Dato che la lista è una sequenza **ordinata** ogni elemento è identificato univocamente dalla sua posizione
- L'operatore di indicizzazione consente di accedere a ogni singolo elemento di una lista, per mezzo dell'indice corrispondente
- **Sintassi**: lista[indice]
  - lista indica il nome di una variabile alla quale sia stata in precedenza assegnata una lista
  - indice deve essere un'espressione il cui valore sia un intero compreso tra 0 e la lunghezza della lista **meno uno**.

Le liste: indicizzazione

- Il risultato dell'operazione di indicizzazione è il valore dell'elemento di lista nella posizione corrispondente al valore di indice. Se il valore di indice non corrisponde a una delle posizioni della lista si otterrà un messaggio d'errore.

- L'operatore di indicizzazione consente anche di **modificare** i singoli elementi di una lista, attraverso un'istruzione di assegnamento.
- **Sintassi**: lista[indice] = espressione
  - dove espressione indica una **qualsiasi** espressione Python
- Percio l'elemento di lista nella posizione corrispondente a indice viene **sostituito** dal valore di espressione.

Le liste: slicing

- L' operatore di slicing in modi simile all'operatore di indicizzazione è basato sull'uso di indici, ma restituisce non solo un elemento della lista ma una lista composta da una sottosequenza della lista originale
- **Sintassi**: lista[indice1:indice2]
  - dove indice1 e indice2 sono espressioni i cui valori devono essere numeri interi compresi tra 0 e la lunghezza di lista.
- Il risultato è una lista composta dagli elementi di lista aventi indici da indice1 a indice2 - 1 (l'elemento all'indice indice2 **non** viene incluso nel risultato).
- Anche l'operatore di slicing consente di **modificare** gli elementi di una lista, attraverso un'istruzione di assegnamento.
- **Sintassi**: lista[indice1:indice2] = lista2
  - dove lista2 indica un'ulteriore lista, che non deve avere necessariamente le stesse dimensioni dello spazio indicizzato dallo porzione selezionata

Le liste: inserimento e cancellazione

- Dato che le dimensioni delle porzioni delle liste non deve coincidere vuol dire che possiamo ingrandire o rimpicciolire la lista
- Possiamo anche assegnare ad una porzione di lista una lista vuota, il che significa cancellare quegli elementi dalla lista, ad esempio con le istruzioni

```
Lista = ['a', 'b', 'c', 'd', 'e', 'f']
Lista[1:3] = []
```

- daremo vita ad una lista con i seguenti elementi `['a', 'd', 'e', 'f']`
- Allo stesso modo possiamo anche assegnare nuovi elementi ad una porzione di lista 'vuota', ad esempio con l'istruzione

```
Lista = ['a', 'd']
Lista[1:1] = ['b', 'c']
```

- daremo vita ad una lista con i seguenti elementi `['a', 'b', 'c', 'd']`

Le liste: concatenazione

- Questo operatore è analogo al corrispondente operatore del tipo di dato stringa.
- **Sintassi:** lista1 + lista2
- La concatenazione produce una **nuova** lista composta dagli elementi di lista1 seguiti da quelli di lista2, disposti nellostesso ordine in cui si trovano nelle due liste.
- Le liste originali **non** vengono modificate.

Le liste: confronto

- Gli operatori == e != consentono di scrivere espressioni **condizionali** (il cui valore sarà True o False) consistenti nel confronto tra due liste.
- **Sintassi**:
  - lista1 == lista2
  - lista1 != lista2
- dove lista1 e lista2 indicano due liste, che sono considerate identiche se sono composte dallo stesso numero di elementi, e se ogni elemento ha valore identico a quello dell'elemento che si trova nella **stessa posizione** nell'altra lista.

Le liste: in e not in

- Questi operatori consentono di scrivere espressioni **condizionali** che hanno lo scopo di verificare se un certo valore sia presente o meno all'interno di una lista.
- **Sintassi**:
  - espressione in lista
  - espressione not in lista
- dove espressione indica una qualsiasi espressione Python e se il **valore** di espressione è presente tra gli elementi di lista l'operatore `in` produce il valore True; in caso contrario produce False. Il comportamento dell'operatore `not in` è quello opposto. La ricerca **non** viene estesa agli elementi di eventuali liste nidificate all'interno di lista.

Le liste: funzioni predefinite

- Alcune funzioni predefinite di utilità generale sono le seguenti:
  - `len(lista)`
    restituisce la lunghezza di una lista
  - `min(lista)`
    restituisce l'elemento più piccolo di una lista composta da numeri
  - `max(lista)`
    restituisce l'elemento più grande di una lista composta da numeri
  - `sum(lista)`
    restituisce la somma di una lista composta da numeri

Le liste: costruzione per concatenazione

- In molti programmi è necessario costruire una lista composta da valori che dovranno essere acquisiti **durante l'esecuzione** degli stessi programmi. La lista non può quindi essere scritta in modo esplicito al loro interno, ma dovrà essere ottenuta come risultato di un'opportuna sequenza di operazioni. Per esempio, è possibile costruire una lista attraverso l'operatore di concatenazione, partendo da una lista vuota.
- Esempio

```
print("Inserire cinque numeri.")
lista = [ ]
k = 1
while k <= 5 :
    elemento = input("Prossimo valore: ")
    lista = lista + [elemento]
    k = k + 1
print("La lista è:", lista)
```

Le liste: funzioni predefinite

- Le liste contenenti numeri sono le più comuni e per questo motivo Python fornisce un modo semplice per crearle
- La classe `range` infatti restituisce un 'oggetto', composto da una sequenza immutabile di numeri, che può essere facilmente trasformato in una lista
- **Sintassi**:
  - range(a)

  se a > 0, crea un oggetto con numeri nell'intervallo [0, a)
  - range(a, b)

  se a e b > 0, crea un oggetto con numeri nell'intervallo [a, b)
  - range(a, b, s)

  se a e b > 0, crea un oggetto con numeri nell'intervallo [a, b) distanziati tra loro di s
  - list(range(..))

  restituisce una lista con numeri nell'intervallo definito da range