Introduzione a Python e accenni alla Programmazione Funzionale

Obiettivi di questa parte del corso

Introdurre la programmazione funzionale e la programmazione in Python.

Perche' la scelta di Python? Non e' un linguaggio principalmente funzionale, ma e' un linguaggio multiparadigma. Ci sono linguaggi principalmente funzionali (e.g. OCaml, Haskell, F#). Abbiamo scelto Python perche' e' molto usato mentre i linguaggi prima menzionati hanno una diffusione piu' ristretta. Attualmente la maggiore parte dei linguaggi include diversi paradigmi di programmazione; sia procedurale che OO che funzionale. Altri vantaggi di Python sono:

- E' `relativamente` facile da imparare
- Mentre Python stesso e' interpretato e quindi meno efficiente, e' stato progettato per potersi interfacciare con librerie scritte in C, quindi molto piu' efficienti.
- Esiste una quantita' enorme di librerie

Perche' la programmazione funzionale?

- Ha sottostante una teoria matematica (lambda calcolo)
- E' molto diversa dal paradigma procedurale, e quindi vi insegna a pensare alla programmazione da un punto di vista molto diverso (e spesso utile)
- Per molti problemi una soluzione funzionale e' la piu' semplice da scrivere
- Si presta alla parallelizzazione automatica dei programmi, cosa molto piu' difficile da fare con i linguaggi procedurali o ad oggetti

Quali sono gli elementi della programmazione funzionale?

- Il programma e' composizione di **funzioni**
- Le funzioni possono essere create e usate come qualsiasi altro oggetto:
  - possono essere passate come parametri ad altre funzioni e
  - ritornate da funzioni.
- In linguaggi puramente funzionali i **dati** (anche quelli strutturati) sono **immutabili** e quindi vengono manipolati attraverso funzioni che non li modificano, ma producono in output nuovi dati. (Come in Java gli oggetti di tipo `String` !)
  - in Python ci sono dati immutabili (i tipi primitivi, stringhe, tuple) e dati mutabili (liste, insiemi, dizionari)
- Si possono definire compuatzioni e dati **lazy**, i cui elementi vengono prodotti solo **se necessario**.

Testi di consultazione

```
- "Programmare con Python Guida completa" di Marco Buttu
- "Functional Python Programming" di Steven Lott
```

Python

- Creato da Guido van Rossum, programmatore olandese: https://it.wikipedia.org/wiki/Guido_van_Rossum (https://it.wikipedia.org/wiki/Guido_van_Rossum)
- Il nome arriva da Monty Python's Flying Circus: https://it.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus (https://it.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus)
- Primo release 1991

- Versione 2.0 release 2000
- Versione 3.0 release 2008
- Versione 3 e' molto diversa dalla versione 2.0, non e' backward-compatible (cioe' molto codice scritto in 2.x NON GIRERA' senza modifiche su un interprete della versione 3)
- Sviluppo della versione 2 e' finito il primo Gennaio 2020

Il software che useremo

Nel laboratorio e' installato Anaconda3, che include:

- Python 3.7.3
- Jupyter Notebooks
- Spyder IDE
- Tante librerie

Noi useremo principalmente `Notebooks e Spyder`

L'applicazione `anaconda-navigator` fornisce un front-end alle altre funzionalita'. Per lanciarlo:

```
/opt/anaconda3/bin/anaconda-navigator
```

Jupyter Notebook

Il contenuto delle lezioni che proiettero' a lezione e' contenuto in files `notebook` (estensione `.ipynb`) che contengono *testo* e *codice*.
Un file notebook e' diviso in **celle**; ogni cella puo'contenere

- testo annotato (in modo simile al `HTML`), cella **Markdown**, oppure
- una finestra per immettere istruzioni Python che si possono eseguire, cella **Code**.

Markdown e' un linguaggio di markup molto semplice. Link https://it.softpython.org/project/markdown.html (https://it.softpython.org/project/markdown.html) a una breve sintesi del linguaggio.

Jupyter Notebooks e' una applicazione client-server. Una volta lanciata apre una pagina in un browser nel quale si possono caricare o creare dei file notebook (estensione `.ipynb`).

Per questo corso **non e' necessario** che sappiate editare un file notebook. Vi faccio solamente una breve introduzione nel caso vogliate usare notebook per prendere appunti (potete creare un vostro file notebook, oppure modificare quello che io vi fornisco.)

Utilizzare Jupyter Notebooks Lanciare Jupyter Notebooks, o dal navigator o da riga di comando. **Jupyter Labs** e' un software piu' recente, evoluzione di Jupyter Notebooks. Guardatelo se volete ma io non l'ho usato.

Una volta lanciato avrete una vista sul vostro filesystem. E' possibile modificare la directory di default, vedere (1). Scaricate da Moodle il file notebook della lezione, e apritelo dentro Notebooks.

Se cliccate su una cella questa diventa la cella corrente e avra' un contorno o verde o blu. Verde indica che siete in modalita' **Edit**, mentre blu indica la modalita' **Command**. In Edit mode, potete eseguire la cella digitando `ctl-enter` . Per una cella Markdown, appare il markup formattato. Per u na cella Code, viene eseguito il codice Python.

NB ci possono essere molte celle Code in un notebook, ma tutte queste condividono lo stesso kernel. Quindi gli oggetti e variabili che si creano eseguendo una cella sono disponibili nelle altre. L'ordine di esecuzione e' importante: se eseguite una cella che fa riferimento a un oggetto creato in una cella precedente, ma quella cella non e' stata eseguita, l'oggetto non sara' definito.

Attenzione, in modalita' Command, ci sono molti shortcut definiti, quindi e' meglio non digitare in modalita' Command (a meno di non sapere bene cosa fanno i comandi). Potreste modificare/cancellare per sbaglio delle celle. Per entrare in modalita' Edit, si puo' o premere `enter` o cliccare nella cella con

```
(1) Per modificare la directory di default di jupiter notebook
bisogna generare e modificare il config file, come segue
eseguire da riga di comando:

>jupyter notebook --generate-config

Questo genera il file

  .jupiter/jupyter_notebook_config.py

nella home directory (il file e' uno script Python!)

Trovate, scommentate  e modificate la seguene riga

# c.NotebookApp.notebook_dir = ''

mettendo fra apici la directory che volete
```

Spyder

Spyder e' una IDE per Python inclusa nella distribuzione Anaconda. Per fare gli esercizi e' consigliabile usarla perche' e' quella che userete all'esame. Comunque siete liberi di l'IDE che volete (Python per Eclipse, Visual Studio, PyCharm che ha una versione freeware) oppure di usare l'interprete da riga di comando.

Spyder puo' essere lanciato da anaconda-navigator o da riga di comando. Quando si lancia, si vedono dei pannelli. A sinistra c'e il pannello Editor, in basso a destra la console IPython, e in alto a destra un pannello che contiene diverse tab per altre viste.

Nel pannello `IPython` si possono digitare direttamente istruzioni Python e avere l'output. Si possono iterare le istruzioni precedenti usando freccia-su e freccia-giu'.

I file nell'Editor possono essere hanno dei comandi utili per l'esecuzione, che si possono vedere nel menu *Run*, dove sono presenti anche gli shortcut.

- `F5` - esegue tutto il file
- `ctl-enter` - esegue la cella corrente
- `F9` - esegue la riga corrente

Se l'interprete non risponde (magari e' in un loop infinito) si puo' interrompere con `ctl-C` . Se anche questo non funziona, si puo' chiudere il pannello di IPython, e crearne uno nuovo. Se fate questo perdete lo stato corrente e dovete ricrearlo.

REPL: Read, Evaluate, Print, Loop

Python, e' un linguaggio **interpretato** (caratteristica dei linguaggi funzionali). Quando valutiamo un programma nell'interprete Python ci troviamo in un REPL (per *read, evaluate, print loop*). Se si inserisce un'espressione, questa viene immediatamente interpretata e il suo risultato scritto. *(NON e' necessario scrivere `print` !)* La cella qua sotto e' una cella di tipo ***Code***. Per esempio, se digitate un numero poi `ctrl-enter`, viene stampato il valore del numero. Se digitate un'espressione poi `ctrl-enter`, viene stampato il risultato dell'espressione. Provatelo ora qui sotto.

In [ ]:
```
23+33
```

Hello World in Python (il piu' in breve!)

Il programma che stampa `Hello World` ora diventa ancora piu' semplice. Basta sapere che:

- Python ha una funzione built-in `print` che stampa i suoi argomenti
- Per scrivere un letterale di stringa si scrive la stringa racchiusa fra doppi apici (o apici semplici).

Quindi il programma diventa:

In [ ]:
```
print("Hello World!")
```

Sintassi di base

- Il ragruppamento di istruzioni (blocco) e' fatto con l'**indentazione** invece che con le parentesi graffe
- Il livello di indentazione dev'essere uniforme in un blocco
- Nei costrutti il fatto che la riga successiva sia l'inizio di un blocco e' segnalato da due punti (:)
- La convenzione per l'indentazione e' 4 spazi
- Alcune versioni di Python non permettono di mischiare spazi e tabspaces, pero' molti editor per Python rimpiazzano tab con spazi
- Python e' **case-sensitive**
- i letterali di stringa possono essere delimitati da `"` o da `'` (cosa utile per poter usare `"` o `'` all'interno della stringa)

In [ ]:
```
# Costrutto if then else. Qui l'else appartiene al secondo 'if'
if False:
    if True:
        print('a')
    else:
        print('b')
print('done')
```

In [ ]:
```
# Costrutto if then else. Qui l'else appartiene al primo 'if'
if False:
    if True:
        print('a')
else:
    print('b')
print('done')
```

In [ ]:
```
# Prima di eseguire il seguente, prova a predire cosa stampera'
if True:
    if True:
        print('a')
    else:
        if True:
            print('b')
    print('c')
    if False:
        print('d')
    else:
        print('e')
else:
    print('f')
```

Commenti, assegnamento e test di uguaglianza

- `#` inizia un commento di riga
- Non ci sono commenti multi-riga
  - L'IDE puo' aiutare a fare commenti multi-riga. Nei notebooks, seleziona delle righe e digita `ctl-/` . In Spyder, seleziona le righe e digita o `ctl-1` oppure `ctl-4`
- Come per Java e C
  - `=` (singolo) e' assegnamento ad un variabile
  - `==` (doppio) e' un test per ugualianza

In [ ]:
```
# Usate ctl-/ per
# commentare queste prime
# tre righe, poi valutate la cella
a = 6    # assegna il valore 5 a a
if a == 5:     # controlla se l'attuale valore dell'oggetto assegnato ad 'a' e
    print("a e' 5")
else:
    print("a non e' 5")
```

Variabili e tipi

- La dichiarazione di una variabile non ne specifica il tipo (Python e' ***tipato dinamicamente***, non staticamente come Java)
- I tipi sono memorizzati con gli oggetti
- Tutte le variabili sono riferimenti ad oggetti
- Se si assegna ad una variabile un oggetto di un tipo diverso da quello riferito precedentemente dalla variabile, non c'e' nessun *cambio di tipo*. Semplicemente la variable ora riferisce ad un oggetto diverso. Quello precedente non e' stato modificato.
- La funzione built-in `type` ritorna il tipo di un oggetto.

In [ ]:
```
foo = 3
bar = foo
print('La variabile foo riferisce ad un oggetto di tipo',type(foo),'con valore
print('La variabile bar riferisce allo stesso oggetto',type(bar),'con valore',b

foo = 3.5
print('La variabile foo riferisce ora ad un oggetto di tipo',type(foo),'con va
print('La variabile bar riferisce sempre all\'oggetto originale',type(bar),'co
```

Alcuni tipi built-in

Prima di elencare alcuni dei tipi, notiamo che Python ha il concetto di **(im)mutabilita'**. Un *valore di tipo immutabile non puo' essere modificato* dopo la sua creazione. Nella programmazione funzionale pura tutti i tipi sono immutabili.

Numeri

Tutti i numeri sono **immutabili**.

- **int**: numeri interi
- **float**: con punto decimale, oppure la notazione mantissa/esponente
- **complex**: usa `j` (non `i`) per la parte immaginaria

Sequenze

- **str:** sequenza di caratteri. Sono **immutabili**. NB non esiste in Python il tipo *carattere*, c'e' solo stringa di lunghezza 1.
- **list:** sequenza di oggetti. Sono **mutabili**. Convenzionalmente gli elementi sono tutti dello stesso tipo, pero' possono non esserlo. Una lista si puo' scrivere come la sequenza dei suoi elementi separati da `,` e racchiusa in parentesi quadre (`[ e ]`).
- **tuple:** sequenza di oggetti. Sono spesso di tipi diversi. Sono **immutabili**, pero' i loro elementi possono essere di un tipo mutabile (ad esempio liste). Una tupla si puo' scrivere come la sequenza dei suoi elementi separati da `,` e racchiusa in parentesi tonde (`( e )`).

Iterare su una sequenza: `for`

Il costrutto `for` permette di iterare sugli elementi di una sequenza eseguendo un blocco di codice per ogni elemento (simile al `foreach` Java che vedremo a breve). La sintassi e' la seguente

```
for <var> in <seq>:
    ...
    ...
```

Il `for` puo' avere una clausola `else`, che viene eseguita se il loop termina normalmente (cioe' senza un `break`)

Altri Tipi

- **boolean:** `True e False`
- **dictionary:** insiemi di coppie chiave-valore
- **set:** insieme in senso matematico (non ci sono ripetizioni).
- **file:** per input/output
- **function:** una funzione e' un tipo di Python, che sara' molto importante per la programmazione funzionale.
- **type:** e' un tipo, applicando la funzione `type` ad un oggetto, viene ritornato un oggetto di tipo `type`.
- **None:** un tipo speciale, indica l'assenza di un valore

Letterali

Un letterale e' la sintassi che un linguaggio accetta per denotare nel programma un oggetto di un tipo. Per esempio, nell'istruzione `a = "foo"`, `"foo"` e' un letterale per una stringa.

int

- Interi possono essere arbitrariamente grandi (come ***BigInteger*** in Java)
- `0x` inizia un letterale per un `int` in esadecimale
- `0o` inizia un letterale per un `int` in ottale (il secondo carattere e' 'o' come Otranto)
- `0b` inizia un letterale per un `int` in binario
- In Python un letterale per un **int** NON PUO' iniziare con uno `0`
- Si puo' anche usare il maiuscolo (`0X` , `0O` , `0B`)

In [ ]:
```
# Create un int molto grande. Moltiplicatelo (operatore '*'') per un altro int
a=99999999999999999999999999999999999999999999999999
a*a
```

Gli operatori numerici principali sono:

- `+` per l'addizione
- `*` per la moltiplicazione
- `/` per la divisione
- `//` per la divisione intera
- `**` per l'elevamento a potenza
- `%` per il modulo

In [ ]:
```
print(8 + 3)
print(8 * 3)
print(8 / 3)
print(8 // 3)
print(8 ** 3)
print(8 % 3)
```

float

Un letterale numerico che contiene un punto decimale diventa un tipo **float**. Se volete zero di tipo **float**, usate `0.0`

Si puo' anche usare un letterale con l'esponente, per esempio `1e100` e `2e-40`

In [ ]:
```
# Create qualche numero float. Convincetevi che un float creato con un
# letterale con l'esponente e' lo stesso di uno creato con il punto decimale
3.5
```

complex

Python ha come tipo built-in i **numeri complessi**. In matematica, la parte immaginaria si indica con un `i` . In Python si indica con un `j` .

Si puo' anche creare un numero complesso con `complex(8,2)`

```
8 + 2j
```

Anche se la parte immaginaria e' zero, e' sempre un numero di tipo complesso

In [ ]:
```
# Create qualche numero di tipo complex, e fate un po' di aritmetica
# Create due numeri complessi che, moltiplicati insieme, risultano un numero co
# con la parte immaginaria 0. Verificate che e' sempre un numero di tipo comple
c1=complex(8,2)
c2=8 + 2j
c3=c1+c2
c4=c1*c2
print(c3,c4)
```

bool

`True` e `False` sono keywords che rappresentano vero e falso

Il concetto e' molto semplice. L'implementazione di Python, pero', puo' causare confusione.

In molti linguaggi, per esempio C, si usano gli **int** diversi da 0 per `True` e `0` per `False`. Python e' compatibile con questa convenzione, quindi `True` delle volte agisce come `1` e `False` come `0`.

Se un'istruzione si aspetta un valore booleano, Python fa di tutto per convertire qualsiasi valore a un booleano, non solo `1` e `0`. Per un **int**, solo `0` e' `Falso`, qualsiasi altro **int** (anche quelli negativi) sono `True`. Si puo' vedere il valore booleano di un espressione con la funzione built-in `bool`

Attenzione: anche se `-5`, nel contesto di un booleano diventa `True`, quando si fa un confronto diretto con `True` non e' uguale.

Truthy e Falsy

Nella letteratura di Python, si trovano le parole ***Truthy*** e ***Falsy*** che riferiscono al fatto che, per esempio, `7` non e' uguale a `True`, pero' in un contesto booleano Python agisce come se fosse `True`. Quindi, il valore `7` e' ***Truthy***

In [ ]:
```
print(7 == False, 7 == True)
if 7:
    print('a')
    if 0.0:
        print('b')
    else:
        print('c')
```

Inversamente in un contesto in cui `False` e' usato come intero agisce come `0` e `True` come `1`. er l'indicizzazione, si potrebbe usare `False` e `True` come `0` e `1`:

In [ ]:
```
# suggerimento: NON FATELO MAI
ls = [1,2,3,4,5]
print(ls[False],ls[True])
```

Per le sequenze ***sequenze**, quelle di lunghezza zero sono ***Falsy***.

Quando si usano gli operatori logici `and` e `or`, Python implementa la ***valutazione corto-circuitata*** simile a `&&` e `||` in C e in Java, ma con qualche peculiarita').

Quando si interpreta `a or b or c`, se la prima espressione valuta a `True`, la seconda non viene valutata e viene ritornato il primo valore che e' ***Truthy***. Se non ci sono valori ***Truthy*** viene ritornato l'ultimo valore.

Quando si interpreta `a and b and c`, se la prima espressione valuta a `False`, la seconda non viene valutata e viene ritornato il primo valore che e' ***Falsy***. Se non ci sono valori ***Falsy*** viene ritornato l'ultimo valore.

Le funzioni `any e all`:

`any` valuta una sequenza di espressioni e ritorna `True` se ce n'e' almeno una ***Truthy***, altrimenti ritorna `False`.

`all` valuta una sequenza di espressioni e ritorna `True` se sono tutte ***Truthy***, altrimenti ritorna `False`.

Sia `any` che `all` sono corto-circuitate. Nota pero' che, a differenza di `and` e `or`, ritornano sempre

In [ ]:
```
# Provate a prevedere cosa ritornano le seguenti espressioni.
# Poi provate se l'avete azzeccato. Scommentando l'espressione
# singola ed eseguendola con ctr enter (attenti all'indentazione
# dell'espressione!!!)

# False or 0 or 7 or True
# True or 7
# 8 or 9

# '' or 8
# 8 or ''
# False or ''
# 0 or False or '' or -5 or 0

False and 7
True and 7
True and 0
True and ''
7 and 8 and 9
8 and '' and 9

all([7,8,9])
any(['',"",0])
```

Esempi Truthy-Falsy

Caricate il file `1_Esempi_Truthy.py` nell' IDE Spyder e proviamo a prevedere il risultato delle chiamate delle funzioni. Poi, premendo F9, potete valutare le righe una per una.

Alcuni costrutti e input

`input()`

La funzione `input()` richiede all'utente di digitare un input. Puo' avere un argomento, la stringa di prompt.

In [ ]:
```
nome = input('Tuo nome: ')
print('Ciao', nome)
```

`if` istruzione

Il costrutto `if` di Python e' piu' strutturato di quello di C o Java. In particolare Per fare un 'else if', si usa la keyword `elif`

In [ ]:
```
char = input('carattere: ')
if char == 'a':
    print('Ancona')
elif char == 'b':
    print('Bologna')
elif char == 'c':
    print('Como')
else:
    print('Spanish Inquisition')
```

`if` espressione condizionale (operatore ternario)

In aggiunta a `if` istruzione c'e' (come in C e Java) un `if` epressione (ternaria) che ha la seguente sintassi

```
<expr-then> if <condizione> else <expr-else>
```

In [ ]:
```
# assegna 5 a x se la condizione e' vera e 6 se e' falsa

x=5 if True else 6

y=5 if False else 6
print("x: " + str(x) + ",y: " +  str(y))
```

`while`

Esegue ripetutamente il blocco di istruzioni che segue fino a che la condizione diventa Falsy

In Python, un `while` puo' avere una clausola `else` , che viene eseguita quando la condizione valuta a Falsy.

Come il linguaggio C, Python ha `break e continue` per i loop (da usare con estrema moderazione!).

In [ ]:
```
# Cosi' il loop non viene interrotto, e esegue l'else.
# Se modificate il valore di `interruptAt` a 1,2 o 3, verra' interrotto e non l

interruptAt = 8
i = 1
while (i <= 7):
    if (i == interruptAt):
        break
    print(i)
    i += 1
# else: print('Abbiamo finito senza interruzione')
```

Assegnamento multiplo

Si puo' assegnare a piu' di una variabile contemporaneamente, l'assegnamento e' fatto simultaneamente a tutte le variabili per cui si possono scambiare valori di variabile senza bisogno di variabili temporanee.

In [ ]:
```
a,b,c = 5,6,7
print(a,b,c)

d,e,f = [8,9,10]
print(d,e,f)
```

In [ ]:
```
# assegnamento multiplo puo' anche essere usato per scambiare due valori

a = 5
b = 6
print(a,b)
a,b = b,a
print(a,b)
```

Definizione di funzioni (uno sguardo iniziale)

La sintassi per definire una funzione e':

```
def <nome>(<argomenti>...):
    ...
    ...
```

Non c'e' una fase di compilazione; la funzione e' definita quando viene eseguito il `def`

Per ritornare un valore si usa la keyword `return` . Se non si ritorna esplicitamente un valore, la funzione ritorna `None`

In [ ]:
```
# Se modificate la condizione dell' if, foo sara' definita in modo diverso

if 0:
    def foo():
        print('footrue')
        return 1
else:
    def foo():
        print('foofalse')
foo()
```

Una breve parentesi sulla differenza fra funzioni e metodi

Una funzione viene chiamata digitando il nome della funzione seguito dagli argomenti fra parentesi. Nel paradigma ad oggetti, si definisce un oggetto che contiene sia dati (variabili) che funzioni (metodi). I metodi si invocano scrivendo l'oggetto seguito da `.` e nome del metodo:

```
class HelloWorld:
    def hello(self):
        print("Hello World")
```

che si invoca cosi':

```
   ciao = HelloWorld()  # crea un oggetto di tipo HelloWorld e lo asseg
na alla variabile ciao
   ciao.hello()         # invoca il metodo hello dell'oggetto ciao
```

Python ha sia funzioni che oggetti e metodi. Posso definire il precedente codice come funzione

```
    def hello():
        print("Hello World")
```

che si invoca cosi'

```
        hello()
```

In [ ]:

```
class HelloWorld:
    def hello(self):
        print("Hello World")

ciao = HelloWorld()
ciao.hello()

def hello():
    print("Hello World")

hello()
```

Valutazione Lazy

E' una strategia per ottimizzare il codice.

In Python se scrivete l'espressione `sum = 1 + 2` e la valutate ottenete l'assegnamento a `sum` di `3`. La valutazione viene fatta immediatamente, e si parla di **Valutazione "Eager"**.

La **Valutazione "Lazy"** non valuta immediatamente l'espressione ma lo fa solo quando e' necessario il risultato, in questo caso quando usate la variabile `sum`.

In Python ci sono funzioni built-in che, quando applicate, non restituiscono un valore, ma producono il loro risultato se questo solo quando questo e' necessario ad altre computazioni. Oggi ne vedremo alcune. Poi vedremo anche come Python permette al programmatore di definire le sue funzioni Lazy.

Funzioni che creano sequenze (iterabili)

`range()`

La funzione `range([inizio],fine,[passo])` crea una sequenza di numeri. In Python 2, ritornava una lista. In Python 3, ritorna un oggetto di tipo `range`. E' un esempio di un `iterable`, dei quali parleremo piu' avanti. Per vedere gli elementi di un range, lo convertiamo in una lista, usando `list`.

In [ ]:
```
print(list())
print(tuple(range(1,7)))
print(list(range(1,7,2)))
print(list(range(7,2)))
print(list(range(0,7,2)))
print(list(range(10,0,-1)))
print(list(range(10)))
print(type(range(10)))
```

Un tipico loop di Java:

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

Si puo' fare in Python utilizzando `range`:

```
for i in range(5):
    print(i)
```

enumerate

Spesso, in un loop che itera su una sequenza, abbiamo bisogno anche dell'indice dell'elemento. Questo si puo' fare o usando un contatore esplicito o in maniera piu' idiomatica usando la funzione `enumerate` che costruisce un insieme di coppie *indice*,*elemento* (si puo' anche specificare un parametro con keyword `start` per alterare il valore del contatore):

In [ ]:
```
wordList = ['The','quick','brown','fox']
ix = 0
for x in wordList:
    print('Parola',ix,"e'",x)
    ix += 1
```

In [ ]:
```
# questo e' piu' elegante:
# list(enumerate(wordList))
for ix,x in enumerate(wordList):
    print('Parola',ix,"e'",x)
```

In [ ]:
```
# iniziamo la numerazione da 2 invece di 0 notate il parametro per parola chiav
for ix,x in enumerate(wordList, start=2):
    print('Parola',ix,"e'",x)
```

zip

La funzione `zip` combina gli elementi di diverse sequenze in una sequenza di tuple. Ha un numero variabile di parameteri. Un esempio:

In [ ]:
```
seq1 = [1,2,3,4,5,6,7]
seq2 = 'abcde'
seq3 = ('alpha','beta','gamma','delta','epsilon')
x = zip(seq1,seq2,seq3)
x
list(x)
# print(list(zip(seq1,seq2,seq3)))
```

vediamo che la funzione `enumerate` e' un caso particolare di `zip`

In [ ]:
```
# Definite la funzione enumerate usando range e zip
# enum(seq2) => [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
# per vedere gli elementi fate come per la funzione range!
def enum(seq2):
    return zip(range(len(seq2)), seq2)
list(enum(seq2) )
```

Specifica dei parametri delle funzioni

Visto che Python non e' tipato staticamente, anche nella definizione di una funzione (come nella dichiarazione di un variabile) non c'e' nessuna informazione sui tipi, ne' per i parametri ne' per il valore di ritorno.

Python offre molta flessibilita' nel passare gli argomenti.

```
def fn(a,b,c)
```

fn ha 3 parametri, possono essere passati o posizionalmente o con keyword

In [ ]:
```
def fn(a,b,c):
    print(a,b,c)

# Le seguenti chiamate sono corrette
fn(1,2,3)
fn(1,c=8,b=3)
fn(c=8,b=3,a=1)

# Le seguenti chiamate causano errori; scommentateli uno per uno per vedere l'e

# fn(1,2)
# fn(1,c=8)
fn(1,b=2,3)
```

Si puo' definire un valore di default per un parametro. Tutti i parametri che hanno un valore di default devono essere **dopo** i parametri senza default.

In [ ]:
```
def fn(a,b=3,c=5):
    print(a,b,c)

# fn(1,2,3)
# fn(1,c=8,b=5)
# fn(c=8,b=3,a=1)

# fn(1,2)

fn(8)
```

Se gli argomenti che volete passare ad una funzione sono `iterable`, li potete passare come argomenti posizionali usando l'operatore `*`. In questo modo potete avere funzioni con un numero variabile di parametri.

In [ ]:
```
# fn(*range(3))
fn(list(range(3)))
```

Se gli argomenti che volete passare ad una funzione sono in un `dict`, li potete passare come argomenti keyword usando l'operatore `**`

In [ ]:
```
coppie = {'b':1, 'c':2, 'a':0}
# fn(**coppie)
fn(*range(3))
```

Quando si definisce una funzione, si possono raccogliere argomenti posizionali aggiuntivi in una tupla usando l'operatore `*`. Questo e' simile ai metodi variadici di Java.

In [ ]:
```
def manyArgs(a,b,*altriPos):
    print(a,b,altriPos)

manyArgs(1,2,3,4,5)
```

Quando si definisce una funzione, si possono raccogliere argomenti keyword aggiuntivi in un `dict` usando l'operatore `**`

In [ ]:
```
def extraKeys(a,b,**altriKeys):
    print(a,b,altriKeys)

extraKeys(1,b=2,c=3,xx=4)
```

3. Esercizi

Caricate il file `3_Esercizi_SD.py` e fate i due esercizi proposti.

Moduli

In Python, ogni file importato crea un nuovo namespace.

`import <nomeDelModulo>`

da' accesso al contenuto del modulo. Deve essere acceduto attraverso il nome del modulo. Proviamo con un modulo dalla libreria standard di Python, `math`

In [ ]:
```
import math
print(math.sqrt(8))
print(math.pi)
```

Si puo' dare un alias al modulo importato

`import <lib> as <alias>`

Nel seguente esempio, importo il modulo `random` , ma se non voglio digitare sempre `random` prima di ogni funzione posso sccegliere un alias piu' corto

In [ ]:
```
import random as ran
ls=[]
for x in range(10):
    ls.append(('testa','croce')[ran.randint(0,1)])
ls
```

Posso importare degli oggetti direttamente nel namespace corrente con la sintassi:

`from <module> import <name>`

`<name>` puo' anche essere una serie di nomi separati da virgola.

Si puo' importare tutto il contenuto di un modulo usando `*` per `<name>`

Nel seguente esempio, importo la funzione `math.sqrt` nel namespace corrente. Cosi' non devo usare il prefisso `math.`

In [ ]:
```
from math import sqrt
sqrt(7)
```

In [ ]:
```
# la radice quadrata di un numero negativo da' errore
sqrt(-1)
```

C'e' un modulo `cmath` , che definisce delle funzioni per l'aritmetica complessa. Esso ha una funzione `sqrt` che da' un risultato per `sqrt(-1)`

In [ ]:
```
import cmath
cmath.sqrt(-1)
```

Posso anche importare la funzione `cmath.sqrt` nel namespace corrente. Sovrascrivera' quella di `math` . Per questo si deve stare attenti a fare questo tipo di `import` (e' proprio per evitare questo tipo di conflitto che esistono i namespace)

In [ ]:
```
from cmath import sqrt
sqrt(-1)

# posso sempre accedere a quella di 'math' usando il namespace
math.sqrt(-1)
```

Qualche Modulo Utile

Vedete https://docs.python.org/3/library/index.html (https://docs.python.org/3/library/index.html) per documentazione completa della Standard Library

Modulo `os`

Informazioni dipendenti dal sistema operativo

In [ ]:
```
import os
# nome del sistema operativo
print('os:',os.name)
# directory corrente
print('cwd:', os.getcwd())
# lista del contenuto di una directory
print(os.listdir(os.getcwd()))
```

Modulo `random`

Generatori di numeri pseudo-casuali

In [ ]:
```
import random
# intero random in range ([start],stop,[step]) default per start e' 0
# step e' la differenza fra 2 numeri consecutivi con default 1

print([random.randrange(5,12,2), random.randrange(5,12,2),random.randrange(5,12

# mescolare una sequenza (modifica la sequenza)
a = list(range(10))
random.shuffle(a)
print(a)

# con seed si possono avere risultati deterministici (utile per debugging)
random.seed(44332)
print([random.randrange(5,20,2), random.randrange(5,20,2),random.randrange(5,12
random.seed(44332)
print([random.randrange(5,20,2), random.randrange(5,20,2),random.randrange(5,12

print([random.randrange(5,20,2), random.randrange(5,20,2),random.randrange(5,12
```

Modulo `re` - espressioni regolari

Modulo `files, os.path,pathlib` - per gestire path e file

Modulo `timeit` - per misurare tempi di esecuzione

Modulo `datetime` - per date e ore

Eccezioni e lettura da file

Le eccezioni (`Exception`) sono un modo di gestire errori. Le vedremo in dettaglio in Java. Ora ne anticipiamo le principali caratteristiche.

- Un'eccezione interrompe il flusso normale del programma.
- Quando succede un'eccezzione, la macchina virtuale di Python cerca un possibile ***handler*** (cioe' manipolatore di eccezioni) per gestirla.
- Se succede un'eccezione e non c'e' nessun handler, il programma esce con un errore.

Quando si esegue una porzione di codice che potrebbe causare un'eccezione e si vuole fornire un handler lo si fa mettendo il codice nel costrutto `try e except` :

In [ ]:
```
# Qui, la divisione per 0 solleva un'eccezione che causa
# l'interruzione dell'esecuzione del programma
def div(n,d):
    return n/d

print(div(2,0))
print(3)
```

Nel costrutto si `try` puo' specificare `finally` che viene eseguito sia che il codice nel blocco `try` provochi un'eccezione che non la provochi. Consideriamo la seguente lettura da file. (Notate che anche i files (e le directory) sono sequenza di linee (di files) per cui si puo' iterare.)

In [ ]:
```
numeri = []
dati = open("WordSquare.txt","r")

try:
    for line in dati:
        numeri.append(float(line))
        print(line)
# except ValueError:
#     print("Errore!!!")
finally:
    dati.close()

print(dati.closed)
```

NB Valgono le considerazioni che abbiamo fatto per la gestion delle eccezioni in Java. Cioe' se non sai cosa fare non le catturare!

L'istruzione `With`

Il pattern precedente cioe' aggiungere una azione che deve essere fatta indipendentemente dal fatto che un certo blocco di istruzioni provochi o meno un'eccezione e' realizzato dall' inziare il blocco che potrebbe causare l'eccezione con l'istruzione `with` . L'espressione nel `with` deve supportare operazioni di `__enter__` and `__exit__` nel caso dei files questo corrisponde a aprire e chiudere i file. Per esempio, leggere un file come nel seguente codice garantisce che il file venga chiuso quando si esce dal `with` .

Il metodo `rstrip()` rimuove i caratteri alla fine della stringa, il default da rimuovere e' il bianco.

In [ ]:
```
numLines = 0
try:
    with open('WordSquare.txt','r') as myFile:
        for line in myFile:
            line = line.rstrip()
            numLines += 1
    print(numLines)
except FileNotFoundError:
    print("File doesn't exist")

print(myFile.closed)
```

Listare i files in una directory

La funzione `walk` del modulo `os` data una directory ritorna una tripla di valori (cammino_corrente, directories nel cammino_corrente, files nel cammino_corrente).

In [ ]:
```
import os
for path,di,files in os.walk('.'):
    for file in files:
      print(file)
```

Sequenze

L'**accesso** agli elementi di una sequenza si fa con la notazione degli array cioe' mettendo l'indice fra parentesi quadre: `aSeq[i]` ritorna l'iesimo elemento della sequenza.

Gli **indici** iniziano da 0: il primo elemento e' `aSeq[0]` Si possono specificare indici negativi e se l'indice e' negativo, si inizia dalla fine della sequenza: `aSeq[-1]` e' l'ultimo elemento. Se `ls= [1,2,3,4]` :

- `1` e' indicizzato da `0` e `-4` ,
- `2` e' indicizzato da `1` e `-3` ,
- `3` e' indicizzato da `2` e `-2` ,
- `4` e' indicizzato da `3` e `-1` ,

Ci sono delle funzionalita' definite per tutte le sequenze.

L'operatore `+` ritorna la concatenazione delle due sequenze. Non altera le sequenze, ne crea una nuova. Le due sequenze devono esserre dello stesso tipo.

L'operatore `*` ritorna `n` ripetizioni della sequenza.

`x in aSeq` ritorna `True` se `x` e' un elemento della sequenza, altrimenti `False` . Si puo' negare con `not (x in aSeq)` oppure `x not in aSeq`

Si puo iterare una sequenza con `for x in aSeq:`

Si puo' usare **slicing** su tutte le sequenze. La sintassi completa e' `aSeq[inizio:fine:passo]` . Vediamo nel seguito dettagli ed esempi.

La funzione built-in `len()` ritorna il numero di elementi nella sequenza.

Le funzioni built-in `min()` e `max()` ritornano il minimo e il massimo elemento della sequenza. E' un errore se ci sono due elementi non comparabili nella sequenza.

Il metodo `aSeq.index(x)` ritorna l'indice della **prima** occorenza di `x` in `aSeq` . Se `x` non c'e', ritorna un errore. Usate `in` prima per evitare l'errore.

Il metodo `aSeq.count(x)` ritorna il numero di volte che l'elemento `x` appare nella sequenza.

Come visto in precedenza si puo' iterare su una sequenza con il costrutto `for` :

```
for <var> in <seq>:
    ...
    ...
```

Stringhe

Le stringhe sono sequenze di caratteri. In Python, non esiste un tipo di dati *carattere*, ci sono solo stringhe di lunghezza 1.

Il tipo str e' immutabile

Ci sono molte varianti di letterali per le stringhe. Quella piu' comune usa un apice singolo: `a = 'foo'`

Come sempre, c'e' il problema: e se voglio un apice singolo nella stringa? Per questo si puo' usare il carattere di escape: `\` . Quindi `a = 'fo\'o'`

Python offre un'altra soluzione: si possono usare i doppi apici per delimitare una stringa. Quindi `a = "fo'o"` . Ovviamente, potrei voler creare una stringa che contiene sia un apice singolo che un doppio apice, a questo punto devo usare il carattere di escape.

Per stringhe multi-line, si puo' usare o `'''` (tre singoli apici) o `"""` (tre doppi apici) come delimitatore:

In [ ]:
```
print('''Questa e' una
        stringa
      su tre righe'''
     )
print("""Lo e'
 anche
   questa"""
     )
```

Il backslash puo' anche essere usato per inserire caratteri speciali, per esempio `\n` per newline o `\t` per tab. Per inserire un `\` , si usa `\\`

stringhe raw

Se la scrittura di una stringa comporta usare molte volte `\\` si puo' usare una **raw string** che inizia con `r` seguito da una stringa, e nella stringa il backslash non e' interpretato come carattere di escape. Ad esempio

In [ ]:
```
r'foo\bar\nerf'
```

f-string (interpolazione)

In Python 3.6 c'e' un modo comodo per formattare una stringa, chiamato **f-strings** che inizia con `f` seguito da una stringa. Questo metodo permette di includere il risultato della valutazione di una espressione dentro un letterale di stringa includendolo fra parentesi graffa aperta `{` e parentesi graffa chiusa `}` . Ad esempio

In [ ]:
```
f'0 or False or  or -5 or 0={0 or False or "" or -5 or 0}'
```

stringhe consecutive

Cosa succede se in una stringa si usano due doppi apici? Cloe' `"foo""bar"` cosa produce? Provatelo

In [ ]:
```
"foo""bar"
```

Perche' questo output? Perche' c'e' un'altra particolarita' sintattica di Python: stringhe consecutive sono concatenate l'una all'altra.

Ora guardiamo alcuni metodi di `str` .

Esempi metodi su stringhe

Per dimostrare alcuni dei metodi piu' comuni per le stringhe caricate il file `2_StringMethods.py` nell' IDE Spyder. Cosi', premendo F9, potete valutare le righe una per una.

NB Le stringhe sono immutabili. Quindi, quando vedete un metodo 'replace' che dice di 'rimpiazzare' elementi di una stringa, sappiate che non modifica nessuna stringa; crea una nuova stringa che corrisponde a quella che ottereste modificando la stringa originale. (Questo e' vero anche per le

`String` di Java)

In [ ]:
```
# Eseguite il loop prima come e' scritto
# Poi scommentate il 'break' e rieseguite

for char in 'foo':
    print(char)
#     break
else:
    print('finito normalmente')
print('dopo il loop')
```

Slicing

Slicing e' un'operazione definita per tutte le sequenze, ma guardiamola utilizzando le stringhe

Fare una slice di un sequenza vuol dire produrre un'altra sequenza (nuova) a partire dagli elementi della sequenza originale.

Per definire uno slice si specifica l'indice della sequenza da cui partire quello a cui fermarsi e infine il passo a cui prendere gli elementi: `aSeq[inizio:fine:passo]` (Ognuno dei parametri e' opzionale ed ha un valore di default.)

L'elemento al primo indice e' **incluso**, e quello all'ultimo e' **escluso**.

In [ ]:
```
a = 'abcdefg'
b=a[1:3:2]  # 'b'
# b[1]
a
```

Ricordiamo che gli indici iniziano da 0. Quindi 1 indica il secondo elemento ('b') e 3 indica il quarto ('d'). Pero' il 'd' non e' incluso, quindi ritorna 'bc'.

- Se non si specifica il `passo` il suo valore e' 1 (e si puo' omettere un `:`)
- Se il `passo` e' positivo e
  - non si specifica l' `inizio`, si inizia dal primo elemento (`inizio=0`),
  - se non si specifica la `fine` si va fino all'ultimo elemento (`fine=len(aSeq)`)

In [ ]:
```
a[:3]
```

Se non si specifica l'ultimo indice, va fino alla fine:

In [ ]:
```
a[1:]
```

Cosi', omettendo entrambi, risulta una copia della stringa

In [ ]:
```
a[:]
```

Abbiamo detto che uno slice viene costruito da una sequenza, ma e' una ***sequenza nuova***. Cosa succede con il codice seguente?

In [ ]:
```
ls=[1,2,3]
x=ls[:]
print(x)
ls[1]='?'
print(ls)
print(x)
```

Ricordate che gli **indici negativi** indicano che partiamo dall'**ultimo elemento che ha indice -1**:

In [ ]:
```
# 'abcdefg' -5 ('c') e' incluse 3 ('c') e' escluso
a
a[-5:2]
```

In [ ]:
```
a[3:-1]
```

Se l'ultimo indice e' uguale o minore del primo, risulta una sequenza vuota

In [ ]:
```
a[3:3]
```

Abbiamo visto che c'e' un terzo elemento nello slice, che e' il `passo` , o step. In questo caso gli elementi vengono prodotti aggiungendo all'indice del precedente il passo.

In [ ]:
```
a[::2]
```

In [ ]:
```
a[1::2]
```

Il `passo` puo' essere **negativo**, in questo caso lo slice **va dal destra a sinistra** perche' aggiungiamo un numero negativo. Il primo indice deve essere maggiore del secondo.

In [ ]:
```
a[7:0:-1]
```

Quando il `passo` e' negativo

- se non si specifica l' `inizio` , si inizia dall'ultimo elemento (`inizio=len(aSeq)`),
- se non si specifica la `fine` si va fino al primo elemento incluso (`fine=-1`)

In [ ]:
```
a[::-1]
```

In [ ]:
```
a[-2::-2]
```

Liste

Il tipo `list` e' uno dei piu' usati in Python. E' una sequenza mutabile. Si indica usando le parentesi quadre.

Si puo' anche creare con la funzione `list` a partire da un iterabile (lo vediamo in seguito!)

In [ ]:
```
# verifichiamo che le funzioni de sequenze funzionano anche per liste
primes = [2,3,5,7,11]
print(len(primes))
print("Primes contiene 7?",7 in primes)
print(primes * 2)
print(primes)
print("Il secondo elemento di primes e'",primes[1])
print("List da stringa",list('foobar'))
```

Liste possono contenere elementi di tipi diversi. Normalmente, pero', sono tutti dello stesso tipo

In [ ]:
```
l2 = [1, 'foo', 3.8, 8j, [1,3]]
for x in l2:
    print(type(x))
```

In [ ]:
```
# Le liste sono mutabili.
primes = [2,3,5,7,11]
primes2 = primes
print(primes, primes2)
primes[0] = 13
print(primes,primes2)
primes.append(17)
print(primes,primes2)
primes2= primes2[:]
primes.extend([19,23])
print(primes,primes2)
```

Si possono selezionare parti di una lista (in generale una sequenza) attraverso l'assegnamento multiplo a variabili. Una delle variabili puo'essere preceduta da `*` e a questa viene assegnata la lista dei restanti elementi.

In [ ]:
```
x1,x2 =[1,2]
print(x1,x2)
# x1,x2,x3 =[1,2,3]
# x = [0,1,2,3,4,5,6,7]
# primo, *altri = x        # primo si lega al primo elemento e altri?
# # print(primo,altri)
# # primo, altri = x        # primo si lega al primo elemento e altri?
# print(primo,altri)
# *primi, ultimo = x
# print(primi,ultimo)
# # a,b,c,*rest,d = x
# # print(a,b,c,rest,d)
# primo,*mezzo,ultimo = x
# print(primo,mezzo,ultimo)

# # Il seguente assegnamento genera un errore
# # primo,*altri,*nonsipuo = x
```

1. Esercizi su stringhe e slicing

Caricate i files `1_Esercizi_String.py` e `3_Esempi_slice.py` e provate a fare gli esercizi proposti.

Assegnamento a una slice

Uno slice ritorna una lista nuova. Pero', un assegnamento diretto a uno slice modifica la lista

In [ ]:
```
a = [1,2,3,4,5]
sl = a[::2]
print('slice prima',sl)
sl[0] = 7
print('slice dopo', sl, 'a dopo assegnazione a sl', a)
a[:3] = [7]*3
print('a dopo assegnazione alla slice', a)
```

Assegnamento ad uno slice (di elementi adiacenti) puo' anche modificare la lunghezza di una lista

In [ ]:
```
a
a[1:3] = [8]
print('a accorciato',a)
a[2:] = [9,10,11]
print('a allungato',a)
```

Per fare l'assegnazione ad uno slice esteso (passo > 1) la lunghezza dello slice e quella della sequenza che si sosituisce devono essere uguali

In [ ]:
```
print('prima',a)
a[::2] = [1,2,3,4] # non puo' essere fatto con [8] * 2 o [8] * 4
print('dopo',a)
```

Tuple

Le tuple sono come le liste, pero' sono **immutabili**. Comunque, un elemento di una tupla puo' essere mutabile.

Le tuple sono spesso usate con elementi eterogenei.

Il letterale per creare una tupla usa le parentesi tonde. Ci sono situazioni nelle quali si possono omettere le parentesi e usare una sequenza di oggetti separati da virgola.

Si possono anche creare tuple con la funzione `tuple`

In [ ]:
```
t1 = (1,2,'three')
print(t1,type(t1))
t2 = 4,5,'six'
print(t2, type(t2))
print('primo elemento di t1:',t1[0])
print('tupla da lista',tuple([1,2,3]))
```

In [ ]:
```
# Il seguente genera un errore, perche' le tuple sono immutabili
t1[0] = 8
```

Piccolo inghippo: non e' ovvio come creare una tupla di lunghezza 1.

Per le liste, non ci sono ambiguita': `list1 = [2]` # crea una lista di lunghezza 1

Pero' le parentesi tonde sono anche usate per raggruppare le espressioni

`foo = (3 + 4) * 5` # si valuta l'espressione e foo sara' un int con valore 60

`tuple1 = (3)` # si valuta l'espressione e tuple1 sara' un int con valore 3

Per evitare cio' si termina la tupla con una virgola.

`tuple1 = (3,)`

o anche

```
tuple1 = 3,
```

Comunque, le parentesi tonde vuote creano una tupla vuota

```
emptyTuple = ()
```

In [ ]:
```
# provate a creare un tuple di lunghezza 1. Controllare il tipo, e la sua lungh
x=(1,)
x
```

Set

Un letterale di tipo `set` (insieme) e' delimitato dai {}

Non puo' avere due elementi uguali.

```
foo = {1,2,3,2,3,1}
```

crea un `set` di 3 elementi

Anche i dizionari (che vedremo fra poco) sono delimitati dai {}.

```
foo = {} # crea un dizionario vuoto, non un insieme vuoto
```

Per creare un `set` vuoto, si deve usare la funzione 'set()'

```
foo = set() # crea un set vuoto
```

`set` ha i seguenti operatori built-in:

- - differenza di insiemi
- | unione di insiemi
- & intersezione di insiemi
- ^ contiene gli elementi che sono in uno dei due insiemi, ma non in entrambi

In [ ]:
```
set1 = {1,2,3,4,5,4,4}
set2 = {3,4,5,6,7}

print(set1 - set2)
print(set2 - set1)
print(set1 | set2)
print(set1 & set2)
print(set1 ^ set2)
```

Dictionary

Un dizionario e' un insieme di coppie chiave-valore.

E' un tipo di dato molto utile e molto usato. La sua implementazione e' fatta in modo tale da rendere veloci le operazioni di aggiungere una nuova coppia chiave-valore, e trovare il valore associato ad una chiave.

Le chiavi devono essere immutabili

Il formato letterale per un dizionario e' una sequenza di coppie separate da virgole, racchiuso in parentesi graffe `{` e `}`. In ogni coppia il carattere `:` separa la chiave dal valore.

In [ ]:
```
dict1 = {'foo': 3, 'bar':4}
dict1
```

In [ ]:
```
# usa keyword 'in' per vedere se una chiave esiste:
print('foo' in dict1)
print('fo' in dict1)
print(3 in dict1)
```

In [ ]:
```
# Usa l'indicizzazione per ottenere un valore.
print(dict1['bar'])
```

In [ ]:
```
# E' un errore se la chiave non esiste
print(dict1['goo'])
```

In [ ]:
```
# per aggiungere una nuova chiave/valore, oppure sovrascrivere il valore di una
dict1['gorp'] = 'yum'
dict1['bar'] = [1,2,3]
dict1
```

In [ ]:
```
# La tupla e' immutabile, quindi puo' essere usata come chiave:
dict1[(1,2,3)] = 'tuple'
dict1
```

In [ ]:
```
# Come gia' detto, la tupla e' immutabile, quindi puo' essere una chiave...
newTuple = (1,2,[3,4])
dict1[newTuple] = 'uhoh'
dict1
a=[3,4]
print(f"a dopo a=[3,4]:{a}")
t=(1,a)
print(f"t dopo t=(1,a):{t}")
a[0]=0
print(f"t dopo a[0]=0:{t}")
```

Cosa e' successo?

Anche se si vede spesso affermato che la chiave di un dizionario Python deve essere immutabile, questo non e' preciso. C'e' un'altro concetto che si chiama ***hashable***. Per essere usato come chiave, un oggetto deve essere hashable. Per essere hashable, deve essere immutabile, e deve contenere solo oggetti immutabili.

`newTuple` sopra e' immutabile, pero' contiene una lista, che e' mutabile. Quindi non puo' essere usata come chiave.

Il termine **hashable** deriva dal fatto che i dizionari sono basati sulle **hash table** (tabelle hash).

In [ ]:
```
tuple2 = (1,2,(3,4))
dict1[tuple2] = 'uhoh'
dict1
```

Per iterare in un dizionario:

```
for x in dict1:
```

Fara' un'iterazione su tutte le **chiavi** di `dict1`

Ci sono anche i seguenti metodi:

```
dict1.keys() # ritorna la sequenza delle chiavi
```

`dict1.values()` # ritorna la sequenza dei valori

`dict1.items()` # ritorna la sequenza delle coppie chiave/valore

Funzioni di conversione

E' importante capire che le **funzioni di conversione di tipo** non cambiano il tipo degli oggetti. In Python, sono funzioni che accettano argomenti di diversi tipi, e cercano di creare un nuovo oggetto del tipo desiderato. Per esempio:

In [ ]:
```
a = '3.3'
b = float(a)
print(b,type(b),a,type(a))
```

Dopo la "conversione", l'oggetto al quale riferisce `a` non e' cambiato. La funzione `float` ha creato un nuovo oggetto.

In [ ]:
```
print(float(True))
print(float(3))
print(float([1]))   # Causerebbe un errore, non sa convertire una lista in un
```

Ecco alcune funzioni Python di conversione

Funzione	Argomenti Validi	Output
int	float, str	int
float	str, int	float
str	int, float, list, tuple, dict	str
list	str, tuple, set, dict	list
tuple	str, list, set	tuple
set	str, list, tuple	set
ord	char	int
hex	int	str
oct	int	str
bin	int	str
chr	int	str

2. Esercizi Strutture Dati

Caricate il file `2_Esercizi_TipiDati.py` e provate a fare gli esercizi proposti.

List comprehension

Python ha una sintassi molto succinta per creare liste a partire da altre sequenze (in generale da `iterable`), che si chiama "list comprehension". La sintassi generale e':

```
[<espressione output> for <variabile> in <iterable> <predicato facoltativo>]
```

In [ ]:
```
[x**2 for x in range(8)]
```

Possiamo usare la clausola `if` per restringere gli elementi enumerati. In questo caso vengono prodotti solo gli elementi che verificano il predicato dopo `if`.

In [ ]:
```
[x**2 for x in range(8) if x%2==1]
```

Si possono anche avere `for` multipli. Come quando si fanno `for` annidati.

In [ ]:
```
[(x,y) for x in range(5) for y in range(5)]
```

Anche in questo caso si puo' usare la clausola `if` per restringere gli elementi enumerati.

In [ ]:
```
#print([(x,y) for x in range(5) for y in range(5) if x>y])
[(x,y) for x in range(5) for y in range(5) if x>y]
```

Iteratori e oggetti iterabili

Come in Java un oggetto iterabile significa che e' in grado di fornire un ***iteratore***. Un iteratore e' un oggetto che permette di scandire uno per uno gli elementi dell'iterabile. Abbiamo gia' visto molti oggetti iterabili. Tutte le sequenze sono iterabili. Anche i file sono iterabili, e' per questo che `for line in myFile` funziona.

E' importante capire che essere iterabile non implica necessariamente che esistano in memoria tutti gli elementi dell'oggetto iterabile. Per le sequenze e' cosi', pero' ci sono molti controesempi. Anche la funzione `range(n)` e' iterabile **ma non crea una lista di elementi**. Una chiamata a `range` ritorna un oggetto di tipo `range`, che e' iterabile.

In [ ]:
```
r = range(10)
print(r,type(r))
z = zip([1,2,3],range(3))
print(z,type(z))
print(list(r))
list(z)
```

Idem per `zip`, che ritorna un iteratore di tipo `zip`. E' per questo che per vedere gli elementi prodotti da un `range` o uno `zip` (nell'esempio sopra) abbiamo usato la funzione `list` che produce una lista.

In Python 3 ci sono molte funzioni che ritornano iteratori: `enumerate()`, `filter()`, `map()`, `reversed()`, `dict.items()`, e molte altre. Vedremo `filter()`, `map()` e altre in seguito.

Perche' un iteratore non ritorna direttamente una lista? Il punto chiave e' che un iteratore non deve necessariamente allocare memoria per tutti gli elementi della struttura su cui si itera contemporaneamente. Questo puo' risparmiare molta memoria. Dato che Python gestisce

automaticamente l'allocazione e deallocazione di memoria dinamica, questo riduce anche il tempo d'esecuzione del programma. In generale, questo concetto concetto si chiama ***lazy evaluation***

4. e 5. Esercizi su comprehension

- Caricate il file `5_Esercizi_SeqCompr.py` e provate a fare gli esercizi proposti.
- Fate l'esercizio contenuto in `4_Esercizio_Words.py` .

Generatori

Python 3 fornisce un modo molto semplice di creare un iteratore (**lazy**) che genera gli elementi che vogliamo. Si puo' creare una funzione particolare che si chiama un **generatore**. La distinzione fra una funzione ed un generatore e' che il generatore invece di ritornare un valore con l'istruzione `return` , lo fa con l'istruzione `yield` . Il punto chiave e' che, dopo un `yield` , la chiamata successiva del del generatore *riprende l'esecuzione subito dopo il yield*, mantenendo tutto il suo stato interno. Vediamo un esempio:

In [ ]:
```
# Una funzione generatore
def mio_gen():
    n = 1
    print('Questo si stampa per primo')
    yield n
    n += 1
    print('Questo si stampa per secondo')
    yield n
    n += 1
    print('Questo si stampa per ultimo')
    yield n

# creiamo un istanza del generatore
a = mio_gen()
print(a)
# # Possiamo iterare sugli elementi usando next().
print(next(a))
# # Quando la funzione raggiunge yields, si ferma e il controllo e' traserito a
# # Le variabili locali sono mantenute da una chiamata all'altra.
print(next(a))
# print(next(a))
# # Quando la funzione termina, le chiamate successive sollevano l'eccezione St
next(a)
next(a)
```

Vediamo un altro esempio, un generatore che produce tutti i caratteri ascii che sono stampabili.

In [ ]:
```
# definiamo il generatore
def printables():
    for i in range(128):
        if chr(i).isprintable():
            yield chr(i)
# creiamo un istanza del generatore
pr = printables()



# # se lo stampiamo, vediamo solo un generatore, non c'e' nessuna lista o seque

print(pr)

# # lo posso usare in una list comprehension (che quindi chiamera' next ripetut

b = [c*3 for c in pr ]  # potete provare a selezionare fra 2 caratteri, ad esem

print( b)

# # il metodo 'join' prende un iterabile, e crea una stringa

print('joined:','.join(pr))

# # perche' non stampa niente? Un generatore puo' essere usato una volta soltan
# # Se voglio iterare di nuovo, devo creare un'altra istanza

pr = printables()
print('joined:','.join(pr))
```

Notiamo che:

1. E' definito come una funzione normale; Python sa che e' un generatore perche' contiene `yield`
2. Dietro le quinte Python ha fatto tutto il necessario perche' possa essere usato come un iterabile
3. Se applichi un iteratore lazy ad altri iteratori lazy, il risultato e' sempre lazy

In [ ]:
```
def talkingRange(*args):
    for i in range(*args):
        print(f'talkingRange: {str(i)}')
        yield i

# range(3) => generatore di 0, 1, 2
# range(3,8,2) => generatore di 3, 5, 7

z = zip(talkingRange(3),talkingRange(3,8,2))
print(z)
# z e' stato creato, ma nessun elemento di talking range e' stato chiamato
# z e' in generatore di (0,3), (1,5), (2,7)
# per produrre gli elementi usiamo il costrutto for e li stampiamo

input('start?')
for p in z:
    print(f"iterazione: {str(p)}")
    input('next?')
```

Generator comprehension

Abbiamo gia' visto l'utilita' di **list comprehension**, che pero' ha lo svantaggio di creare una lista, cioe' non c'e' **lazy evaluation**, ma tutti gli elementi sono creati in memoria.

Python 3 fornisce un altro tipo di comprehension, un generator comprehension. La sintassi e' la stessa della list comprehension, eccetto che e' racchiusa da parentesi tonde invece che quadre. E invece di una lista, ritorna un generatore.

In [ ]:
```
# Cambiando le parentesi alla list comprehension che abbiamo visto:
print([x**2 for x in range(8)])

print((x**2 for x in range(8)))

for x in (x**2 for x in range(8)):
    print(x)
```

6. Esercizi su generatori

Caricate il file `6_Esercizi_Generatori.py` e provate a fare gli esercizi proposti.

Funzioni

Uno degli aspetti di Python che lo rende adatto per la programmazione funzionale e' che **le funzioni sono oggetti di prima classe**. Possono essere create e usate come qualsiasi altro oggetto. In particolare, possono essere passate come parametri ad altre funzioni e ritornate da funzioni.

Ragioniamo con le funzioni

Il concetto di usare funzioni come parametri e valori di ritorno e' in genere difficile da capire, ma gran parte della flessibilita' ed eleganza del paradigma di programmazione funzionale deriva da fatto che le funzioni sono **first-class objects**, cioe' possono essere usate come gli altri tipi di dato.

Che cos'e' una funzione?

Una funziona matematica e' diversa da una funzione in un linguaggio di programmazione (LP), comunque questi concetti sono correlati e quindi conviene ripassare prima cos'e' una funzione matematica.

In matematica, una funzione puo' essere vista come una 'macchina' che trasforma un input in un output. Si scrive:

```
f(x) = y
```

Si dice "f di x e' y", e significa che la funzione `f`, dato l'input `x`, produce il valore `y`.

Si deve specificare quali tipi di input accetta e quale tipo di output produce.

Per esempio, la funzione 'modulo 3' converte un `int` in un altro `int'.

```
f(x) = x % 3
```

La funzione 'quadrato' converte un qualsiasi numero reale in un altro numero reale.

```
f(x) = x * x
```

La funzione 'radice quadrata' converte un qualsiasi numero positivo in un altro numero positivo.

```
f(x) = √x
```

(Abbiamo fatto qualche semplificazione: ignoriamo i numeri complessi, e il fatto che radice quadrata puo' produrre due valori)

Gli esempi sopra sono tutti esempi di funzioni di una variabile. Una funzione puo' avere piu' di una variabile.

```
f(x1,y1,x2,y2) = (x2-x1)**2 + (y2-y1)**2
```

Differenze fra funzioni matematiche e funzioni nei LP

Le funzioni matematiche sono deterministiche

Una funzione matematica e' *deterministica*, cioe' dato gli stessi input produce **sempre** lo stesso output. Quindi, una funzione senza parametri deve per forza essere una funzione costante, cioe' ritorna sempre lo stesso valore.

Le funzioni nei LP sono un po' diverse. Anch'esse hanno un numero di parametri di input. In alcuni linguaggi non e' obbligatorio che producano un output; si possono chiamare per ottenere effetti collaterali. Non e' detto che siano deterministiche; per esempio se abbiamo aperto un file ogni chiamata a `nextline()` produce un valore diverso ad ogni chiamata.

Le funzioni matematiche non possono riferire a variabili esterne

Non sarebbe sensato in matematica scrivere:

```
f(x,y) = x*2 + y*2 + z*2
```

Se la funzione `f` dipendesse anche dal valore di `z`, bisognerebbe scrivere:

```
f(x,y,z) = x*2 + y*2 + z*2
```

Nella maggiore parte dei LP, si possono riferire variabili non locali. Questo e' uno dei modi in cui una funzione puo' diventare non-deterministica: il risultato puo' cambiare se la variabile non-locale cambia. E' per questo che nel paradigma funzionale si cerca di evitare cio', e si prediligono funzioni **pure** cioe' **funzioni che non riferiscono a variabili globali**.

Differenza fra una funzione, e l'applicazione di una funzione

Siamo abituati a pensare alle funzioni come una cosa diversa dai dati; una funzione opera sui dati. Pero', in un linguaggio funzionale, anche le funzioni possono essere dati. Quindi, una funzione puo' essere usata in due modi diversi:

1. Puo' essere invocata, cioe' usata come 'macchina' attiva per operare sui dati
2. Puo' essere usata in modo passivo come un qualsiasi dato

Per non confondersi, bisogna capire quando e' invocata e quando e' usata come dato. In Python, per

In [ ]:
```
def addTwo(x):
    return x + 2
# Ora addTwo e' una variabile, il cui valore e' una funzione
```

In Python, come nella maggiore parte dei linguaggi, gli argomenti sono valutati prima di valutare il body della funzione

Sotto, la variable `addTwo` viene *usata*, il suo valore e' una funzione, che e' l'argomento della funzione `print`

In [ ]:
```
print(addTwo)
```

Sotto, l'espressione `addTwo(8)` viene *valutata*. Questa e' l'applicazione della funzione `addTwo` al suo argomento `8`. Il risultato e' 10, che e' l'argomento della funzione `print`

In [ ]:
```
print(addTwo(8))
```

Scriviamo qualche funzione che accetta una funzione come parametro.

Scriviamo una funzione `applica` che accetta una funzione di un parametro, e la applica ai numeri 5,6 e 7 e ritorna una lista dei risultati.

In [ ]:
```
def applica(fn):
    return [fn(5), fn(6), fn(7)]

# scriviamo le funzioni 'raddoppia' e 'triplica'
def raddoppia(x):
    return x * 2

def triplica(y):
    return y * 3

# Chiamiamo la funzione 'applica' con argomento 'raddoppia' e stampiamo il risu

print(applica(raddoppia))

# idem per 'triplica'

# print(applica(triplica))
```

Scriviamo una funziona 'applica2' che prende una funziona e un numero, e applica la funzione al numero e poi al risultato.

In [ ]:
```
def applica2(fn, arg):
    primaApplicazione = fn(arg)
    secondaApplicazione = fn(primaApplicazione)
    return secondaApplicazione

print(applica2(raddoppia,3))
```

Si puo' scrivere `applica2` in modo piu' succinto (senza definire delle variabili intermedie):

In [ ]:
```
def applica2v2(fn,arg):
    return fn(fn(arg))

print(applica2v2(raddoppia,3))
```

Vedremo che la stessa funzione si puo' scrivere usando la notazione lambda per Python (che in seguito ved remo anche in Java) senza dover definire una funzione con un nome.

Scriviamo una funziona che crea e ritorna una funzione

Abbiamo visto sopra le due funzioni `raddoppia` che moltiplica il suo argomento per 2 e `triplica` che moltiplica il suo argomento per 3. Scriviamo una funzione che prende come argomento un numero `n`, e ritorna una *funzione di un argomento* che moltiplica il suo argomento per `n`:

In [ ]:
```
def creaMolt(n):
    def moltiplicatore(x):
        return x * n
    return moltiplicatore

per4 = creaMolt(4)
print(per4)
per7 = creaMolt(7)
print(per7)

print(per4(1),per4(3))

print(per7(1),per7(3))
```

Lambda notazione

Molto spesso, si vuole definire una funzione che usa una volta soltanto, ad esempio come argomento di un'altra funzione. In questi casi non e' necessario definire un nome per la funzione, perche' questa e' usata solo nel punto in cui e' definita. Python (come abbiamo visto anche in Java) offre un'altra sintassi per facilitare la definizione delle funzione che si chiama espressione `lambda` .

L'uso del termine `lambda` deriva dal `lambda-calcolo` , una teoria della computazione basata sulla nozione di funzione, il 'lambda calcolo'. In Python come in Java `lambda` e' sempliciamente una sintassi per definire velocemente delle funzioni. La sintassi e' la seguente:

```
lambda <parametri>: espressione
```

Cioe', la parola `lambda` , seguita da una lista facoltativa di parametri, seguita da due punti ':', seguiti da un'espressione. Questo ritorna una funzione che ritorna il valore dell'espressione (dopo il `:`) in cui i parametri sono sosituiti dagli argomenti.

Infatti, scrivere:

```
variableName = lambda x: x + 3
```

e' equivalente a:

```
def variableName(x)
    return x + 3
```

In [ ]:
```
# definisco una lambda con zero parametri e' lo assegno al variabile noPar
noPar = lambda: 8
print(noPar)
print(noPar())

# definisco una lambda con un parametro
add3 = lambda x: x + 3
print(add3(7))

#definisco una lambda con due parametri
modSum = lambda x,y: (x + y) % x
print(modSum(3,5))
```

Facciamo un esempio di uso di una lambda come argomento di una funzione.

Consideriamo il problema di ordinare una lista. Un possibile modo e' usare la funzione `sorted` che ordina una sequenza di elementi.

Guardiamo la documentazione di `sorted`

In [ ]:
```
help(sorted)
```

I parametri `key` e `reverse` devono essere passati usando la parola chiave (cioe' il nome del parametro) e specificano rispetto a cosa si deve ordinare e se gli elementi devono essere ordinati in modo crescente (il default) o decrescente.

La lista che vogliamo ordinare contiene coppie con primo elemento il nome di una persona famosa e secondo la sua eta'. Vogliamo ordinare per eta' in modo decrescente. Per fare questo passiamo come secondo parametro una lambda che prende una tupla e restituisce la sua seconda componente. Poi passiamo anche come terzo parametro (ancora per parola chiave) `True` .

In [ ]:
```
tlist = [('Mila Kunis',41),('Mila Kunis',40),('Bono',59),('Eminem',47),('Jack B
tlist
```

In [ ]:
```
sortedList=sorted(tlist)
print("Lista originale\n",tlist)
print("Lista ordinata\n",sortedList)
```

Esiste anche un metodo `sort` della classe lista che si puo' usare nello stesso modo (ma non ha il primo parametro!). In questo caso la lista a cui si applica viene modificata.

In [ ]:
```
print("Lista originale\n",tlist)
tlist.sort()
print("Lista ordinata\n",tlist)
```

Per il parametro `key` si puo' specificare quale funzione viene applicata agli elementi che ci dice rispetto a cosa ordinare. Ad esempio `key=lambda el: el[1]` ci permette di ordinare rispetto alla seconda componente della tuple.

In [ ]:
```
print("Lista originale\n",tlist)
tlist.sort(key=lambda el: el[1])
print("Lista ordinata\n",tlist)
```

Lambda come argomenti di funzioni

Rivediamo gli esempi che abbiamo appena fatto e invece di definire funzioni passiamo delle lambda

In [ ]:
```
def applica(fn):
    return [fn(5), fn(6), fn(7)]

# se usiamo una lambda non dobbiamo definire la funzione

print(applica(lambda x: x - 4))

def applica2(fn, arg):
    primaApplicazione = fn(arg)
    secondaApplicazione = fn(primaApplicazione)
    return secondaApplicazione


print(applica2(lambda x: x - 3, 4))
```

Lambda ritornate da funzioni

Questo vale anche quando ritorniamo funzioni

In [ ]:
```
# possiamo resituire direttamente una lambda

def creaMolt(n):
    return lambda x: x * n

per4 = creaMolt(4)
per7 = creaMolt(7)

print(per4(1),per4(3))
print(per7(1),per7(3))
print(per7(per4(1)))
```

7. Esercizi su funzioni

Caricate il file `7_Esercizi_Funzioni.py` e provate a fare gli esercizi proposti.

Esempi di funzioni che ritornano funzioni fra le funzioni Python che abbiamo visto

L'esempio sopra di `creaMolt` e' facile da capire e serve per dimostrare come creare e ritornare una funzione da una funzione. Pero' non sembra molto utile. E' una tecnica veramente utile?

Basta pensare a tante funzioni che abbiamo usato programmando in Python. Tutte le funzioni come `range`, `enumerate`, `zip`, ecc. creano e ritornano funzioni.

Funzioni built-in che accettono funzioni come parametri

In aggiunta a `sorted` anche le funzioni `min` e `max` che abbiamo gia' visto, per trovare l'elemento minimo e l'elemento massimo di una sequenza possono avere funzioni come parametri.

In [ ]:
```
list1 = [3,1,6,4,5,2]
list2 = ["Peter","Paul","Mary"]
print('mins:\t',min(list1),min(list2))
print('maxes:\t',max(list1),max(list2))
```

Pero' cosa facciamo se gli elementi della lista non sono oggetti semplici? Come esempio la lista di tuple che rappresenta delle celebrita' e le loro eta'. Potremmo voler trovare il max o il min rispetto all'ordinamento dei nomi, o a quello dell'eta'.

Sia `min` che `max` accettano (come `sorted`) un parametro keyword `key` che deve essere una funzione che accetta un parametro e ritorna un valore. Questa funzione viene applicata ad ogni elemento, e il valore ritornato viene usato per calcolare il minimo o massimo.

Per l'esempio sopra, possiamo definire due funzioni, `getFirst` che ritorna il primo elemento della tupla, e `getSecond` che ritorna il secondo elemento della tupla.

In [ ]:
```
tlist = [('Mila Kunis',41),('Mila Kunis',40),('Bono',59),('Eminem',47),('Jack B

def getFirst(tup):
    return tup[0]

def getSecond(tup):
    return tup[1]


print(min(tlist,key = getFirst),max(tlist,key = getFirst))
print(min(tlist,key = getSecond),max(tlist,key = getSecond))
```

Anche la funzione `sorted` e il metodo `sort`, come abbiamo visto accettano questo stesso parametro.

Quando gli oggetti nelle sequenze sono loro stessi sequenze, per loro si considera l'**ordine lessicografico**. Questo vuole dire che si ordina per il primo elemento; se questi sono uguali, si ordina per il secondo; se anche questi sono uguali, si ordina per il terzo, ecc. Questo e' l'ordine che viene usato per le stringhe nei dizionari.

In [ ]:
```
wordList = ["contusione","confusione","aaron","aardvark"]
dateList = [(1972,8,9),(1972,11,9),(1492,8,3),(1969,8,15),(1969,7,21),(1969,7,2
# ordinare parole
print(sorted(wordList))
# ordinare date nel formatto (anno, mese, giorno)
print(sorted(dateList))
```

Come possiamo fare se vogliamo ordinare una lista di date in modo diverso? Diciamo che sono nell'ordine (mese, giorno, anno) e noi le vogliamo ordinare in ordine (anno,mese,giorno). Per fare questo possimo creare una funzione che ritorna una tupla ordinata nel modo che vogliamo:

In [ ]:
```
mgaList = [(8,9,1972),(11,9,1972),(8,3,1492),(8,15,1969),(7,21,1969),(7,20,1969

def dateTuple(mgaDate):
    return (mgaDate[2],mgaDate[0],mgaDate[1])

print(sorted(mgaList, key = dateTuple))
print(sorted(mgaList))
```

Negli esempi sopra, le funzioni `key` erano molto semplici e chiamavano un operatore Python. Per facilitare l'utilizzo di questo tipo di funzione, la Standard Library di Python ha un modulo `operator` che ha gia' definite delle funzioni che corrispondono a molti deli operatori di Python.

L'operatore che abbiamo usato sopra e' `[]`, l'operatore di indicizzazione. Il modulo fornisce una funzione `itemgetter` che ritorna una funzione per acceddere agli elementi del suo argomento.

In [ ]:
```
import operator
getFirstElement = operator.itemgetter(0)
getSecondElement = operator.itemgetter(1)
print(min(tlist, key = getFirstElement))

# cosi', si puo' anche evitare di definire la funzione
print(sorted(tlist, key = operator.itemgetter(1)))

# Se si forniscono multipli argomenti a itemgetter, questa ritornera' una tupla
sorted(mgaList, key = operator.itemgetter(2,0,1))
```

8. Esercizi su min max e sorted

Caricate il file `8_Esercizi_min_max_sort.py` e provate a fare gli esercizi proposti.

Stile Funzionale di Programmazione su Sequenze

Reduce

La funzione `reduce` implementa un tecnica comunemente chiamata *folding* o *reduction*, che consiste nel produrre da una sequenza un valore comulativo (ad esempio la somma dei suoi elementi!). La funzione `reduce` di Python si applica ad un iterabile eseguendo i seguenti passi:

- applicare una funzione `f` ai primi 2 elementi generando un risultato parziale
- usare quel risultato parziale, insieme con un terzo elemento dell'iterabile per generare un altro risultato parziale
- ripetere il processo finche' l'iterabile non e' esaurito e quindi ritornare il valore finale La funzione `reduce` e' nel modulo `functools`

In [ ]:
```
from functools import reduce
help(reduce)
```

La somma degli elementi di una sequenza puo' essere fatto usando `reduce` lo avete visto negli esercizi.

In [ ]:
```
import operator
reduce(operator.add, range(100))
```

Cosa succede se applichiamo la funzione ad una lista con un solo elemento?

In [ ]:
```
# Cosa succede se applichiamo la funzione ad una lista con un solo elemento?

reduce(operator.add, [1])

# e ad una lista vuota?

reduce(operator.add, [])
```

Per queso e' importante fornire anche l'input opzionale `initial` che e' usato come primo valore ad dare alla funzione.

In [ ]:
```
reduce(operator.add, [], 0)
```

Comunque sapete che l'operazione precedente e' fatta dalla funzione predefinita di Python `sum` cioe'

In [ ]:
```
sum(range(100))
```

Notate che la funzione `reduce` coincide con la `foldLeft` degli esercizi!!!!

Performance

Vogliamo misuarare il tempo di esecuzione delle soluzioni precedenti per la somma degli elementi di un iterabile. Per questo possiamo usare la funzione `timeit()` del modulo `timeit` :

```
timeit.timeit(stmt='pass', setup='pass', number=1000000, globals=None)
```

Gli argomenti:

- `stmt` sono le istruzioni di cui volgiamo sapere il tempo di esecuzione
- `setup` sono istrizioni da eseguire prima di `stt` , ad esempio `import` di moduli
- `globals` e' un dizionarion che contiene i nomi definite nell'ambiente che servono per valutare `stmt` Vediamo ora come usarla

In [ ]:
```
from functools import reduce
from timeit import timeit


def add(a, b):
    return a + b

use_add = "functools.reduce(add, range(100))"
print("reduce with use_add:"+str(timeit(use_add, "import functools", globals={'


# Using a lambda expression
use_lambda = "functools.reduce(lambda x, y: x + y, range(100))"
print("reduce with lambda:"+str(timeit(use_lambda, "import functools")))


# Using operator.add()
use_operator_add = "functools.reduce(operator.add, range(100))"
print("reduce with operator.add:"+str(timeit(use_operator_add, "import functool


# Using sum()
print("primitive sum:"+str(timeit("sum(range(100))", globals={"sum": sum})))
```

Map e Filter

Funzioni che prendono funzioni come parametri, oppure che ritornano funzioni come parametri, si chiamano **funzioni di ordine superiore (higher order functions)**. Abbiamo visto alcune funzioni di ordina superiore, ora introduciamo quelle che sono le funzioni di ordine superiore fondamentali nella manipolazione di iterabili (sequenze).

Il paradigma di programmazione funzionale fa frequente uso delle funzioni `map` e `filter` .

`map` prende due argomenti: una funzione e un iterabile. Ritorna un iterabile con la funzione applicata ad ogni membro del secondo parametro.

`filter` prende due argomenti: una funzione e un iterabile. Applica la funzione ad ogni membro dell'iterabile, e ritorna un iterabile contenente solo i valori per cui la funzione ritorna un valore truthy.

In [ ]:
```
list1=(1,2,-3,4,-4,2,4,-23)

# filtriamo list1 ritornando i valori positivi
onlyPos = filter(lambda x: x >= 0,list1)
onlyPos
print(list(onlyPos))

print(list(map(lambda x:x%2==1,list1)))


print(list(map(lambda x: x**3,list1)))

# possiamo combinare map e filter
cubiOdd = map(lambda x: x**3, filter(lambda x: x % 2 == 1, list1))
print(list(cubiOdd))
```

Comprehension e' una sintassi per map e filter

Se gli esempi sopra sembrano familiari, e' perche' list comprehension e generator comprehension che abbiamo visto sono essenzialmente una sintassi di comodo per l'applicazione di map e filter. Quindi quando usiamo la comprehension programmiamo gia' in modo funzionale, cioe' applicando trasformazioni/funzioni a dati.

In [ ]:
```
cubiOdd2 = [x**3 for x in list1 if x % 2 == 1]
print(list(cubiOdd2))
```

Come traduciamo `[ expr  for x in iterable if cond]` con `map e filter` ?

```
list(map(lambda x:expr,filter(lambda x:cond, iterable)))
```

In [ ]:
```
cubiOdd3 = map(lambda x: x**3, filter(lambda x:x % 2 == 1, list1))
print(list(cubiOdd3))
```

9. Esercizi su reduce map filter

Caricate il file `9_Esercizi_reduce_map_filter.py` e provate a fare gli esercizi proposti.

Ricorsione

Un'altra tecnica molto diffusa nel paradigma di programmazione funzionale e' la ricorsione, cioe' una funzione che chiama se stessa (o direttamente o indirattemente). Un esempio classico e' la funzione fattoriale.

In [ ]:
```
def fatt(n):
    if (n <= 0): return 1
    return n * fatt(n-1)

print(fatt(0), fatt(4),fatt(19))
```

Di default, lo stack di ricorsione di Python non può superare i 1000 frame

In [ ]:
```
# Python limita la dimensione dello stack, quindi questa non e' una soluzione v
fatt(3000)
```

Notiamo che e' importante fermare la ricorsione! Tipicamente questo e' il primo controllo in una funzione ricorsiva, perche' si deve fare prima della chiamata ricorsiva. Altrimenti avrai una ricorsione infinita e sicuramente uno stack overflow.

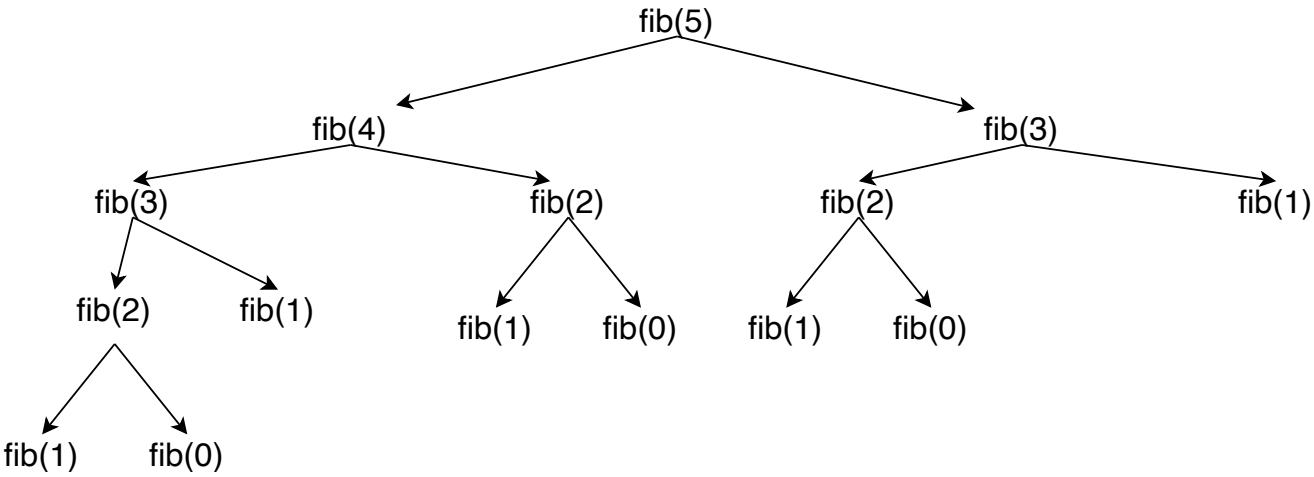
Facciamo un altro esempio: il calcolo dell'ennesimo numero di Fibonacci:

In [ ]:
```
def fib(n):
    if n <= 1: return 1
    return (fib(n-1) + fib(n-2))

list(map(fib,range(10)))
```

In [ ]:
```
# gia' per n = 36 diventa piuttosto lenta
fib(36)
```

La lentezza non e' attribuibile alla profondita' dello stack, ma all'inefficienza della ricorsione. Ricalcoliamo molte volte fib(m).



fib(1) viene calcolato 5 volte, fib(2) viene calcolato 3 volte,.....

Possiamo usare un trucco per migliorarne l'efficienza. Riscriviamo la funzione in modo tale che ritorni non solo l'ennesimo numero di Fibonacci, ma anche quello precedente. Cosi' facendo, riusciamo ad avere una sola chiamata ricorsiva. Quindi la nostra funzione fib2(n) ritorna una coppia (p,q) in cui q e' fib(n) e p e' fib(n-1)

In [ ]:
```
def fib2(n):
    if (n <= 0): return (0,1)
    rec = fib2(n-1)
    return (rec[1],rec[0]+rec[1])


print(fib2(36))
print(fib2(100))
```

Ricorsione di coda

Un particolare tipo di ricorsione, importante nella programmazione funzionale, e' la **ricorsione di coda**. questa ricorsione e' caratterizzata dal fatto che le chiamate ricorsive sono in **posizione di coda** cioe' le espressione ritornate dalla funzione. Le nostre funzioni `fatt` , `fib` e `fib2` *non sono ricorsive di coda*. Infatti la chiamata ricorsiva in `fatt` , `n * fatt(n-1)` , non e' in posizione di coda, come pure

quelle in `fib` e `fib2`. La ricorsione di coda e' molto importante nella programmazione funzionale, perche' puo' essere strasformata in iterazione e quindi non e' necessario allocare record di attivazione per la sua esecuzione. Vediamo come si scrive una funzione ricorsiva di coda per il fattoriale.

In [ ]:
```
def fatt(n):
    def fattRC (n,acc):
        if n<=0:return acc
        return fattRC(n-1,n*acc)
    return fattRC(n,1)

print(fatt(6))
fatt(3000)
```

Notate l'uso della funzione ausiliaria `fattRC` che prende un parametro sul quale si accumulano i risultati parziali. La chiamata ricorsiva `fattRC(n-1,n*acc)` e' in posizione di coda, poiche' il suo risultato viene restituito dalla funzione `fattRC` per cui `fattRC` *e' ricorsiva di coda*.

Purtroppo in Python la ricorsione di coda non viene ottimizzata, per cui e' meglio scrivere direttamente le versioni iterative delle funzioni.

In [ ]:
```
def fattIt(n):
    acc = 1
    while (n>0):
        acc = n*acc
        n = n-1
    return acc

print(fattIt(6))
print(fattIt(3000))
```

E se volessi definire `fib2` ricorsiva di coda?

Un altro tipo di dato: namedtuple (simile a `struct`)

Python fornisce, in un package 'collection', la possibilita' di creare tuple con nomi. In realta' crea un oggetto, con un costruttore e dei selettori.

L'oggetto che crea e' un sottotipo di tupla. Quindi, si puo' sempre trattarlo come una tupla. In piu', viene fornito un costruttore che si puo' chiamare con parametri posizionali o per keyword. Per accedere ai campi si usano i nomi dei parametri.

In [ ]:
```
from collections import namedtuple
Persona = namedtuple('Persona',['nome','eta','qi'])
fred = Persona('Fred',20,100)
sue = Persona('Susan',18,110)
print(fred, sue)
print(fred.qi,sue.qi)
print(fred[2],sue[2])
```

Valori di default per il costruttore possono essere specificati col parametro `defaults`

In [ ]:
```
Persona = namedtuple('Persona',['nome','eta','qi'], defaults=['',0,100])
jane = Persona('Jane')
bob = Persona('Bob',18)
print(jane, bob)
```

Ritorniamo alla ricorsione

La vera utilita' della ricorsione si vede quando si devono processare strutture dati che sono inerentamente ricorsive.

Una sequenza e' una struttura ricorsiva; se non e' vuota e' un elemento seguito da una sequenza. Usiamo questo fatto per scrivere qualche funzione ricorsiva su sequenze.

La seguente funzione calcola la somma delle lunghezze degli elementi in una sequenza

In [ ]:
```
def sumLen(seq):
    # la ricorsione si ferma se la sequenza e' vuota
    if not seq: return 0
    # altrimenti facciamo la ricorsione
    first, *rest = seq
    return len(first) + sumLen(rest)

print(sumLen(()))
print(sumLen(('abc')))
print(sumLen('abcde'))
print(sumLen(['abc','def','ghi','lmnop']))
print(sumLen([(1,2,3),(),(8,),"Phred",[(),(),()]]))
```

Come avrete visto nel corso di Algoritmi, un esempio molto comune di struttura ricorsiva e' un **albero** (tree). Un albero e' un insieme di **nodi**. Ogni **nodo** (node) puo' contenere un dato, e in piu' puo' avere uno o piu' sottoalberi, ognuno dei quali e' un albero. Un nodo senza sottoalberi si chiama **foglia** (leaf). Ogni albero ha un nodo distinto che si chiama **radice** (root). I sottoalberi di un nodo si chiamano **figli** (children).

Il diagramma sopra che traccia il calcolo dei numeri di Fibonacci e' un esempio di albero.

Un **albero binario** (binary tree) e' uno in cui ogni nodo ha un massimo di due figli.

Facciamo un esempio di alberi binari. Definiamo un nodo che ha tre valori: un dato e due sottoalberi.

In [ ]:
```
from collections import namedtuple

Tree = namedtuple('Tree',['data','sx','dx'], defaults=[None]*3)

tree1 = Tree('root')
print(tree1)
tree2 = Tree('+',
             Tree(7),
             None)
print(tree2)
```

La **profondita'** (depth) di un albero e' la distanza massima dalla radice ad una foglia. Come calcolarla? Senza usare la ricorsione e' molto complicato. Con la ricorsione e' invece molto semplice.

In [ ]:
```
# una funzione utile che ritorna True se il nodo e' una foglia
def isLeaf(tree):
    return tree.sx == None and tree.dx == None

# una funzione per calcolare la profondita'
def getDepth(tree):
    if isLeaf(tree): return 0
    return 1 + max([getDepth(subtree) for subtree in tree[1:] if subtree])

# proviamo
print(getDepth(tree1))
print(getDepth(tree2))
```

Creiamo dei alberi che rappresentano dei calcoli con operatori binari. In questi alberi, ogni nodo rappresenta:

1. Un'operazione con due operandi (il dato contiene l'operazione, e i figli gli operandi), oppure
2. Un numero (il dato contiene il numero, ed e' una foglia)

In [ ]:
```
optree1 = Tree('*',
               Tree('+',
                    Tree(7),
                    Tree(4)),
               Tree('-',
                    Tree(7),
                    Tree(4))
               )

optree2 = Tree('*',
               Tree('*',
                    Tree(3),
                    Tree(3)),
               Tree('-',
                    Tree(2),
                    Tree(4))
               )

optree3 = Tree('+',optree1,optree2)

print(optree1)
print(optree2)
print(optree3)
```

Scriviamo una funzione per calcolare il valore.

In [ ]:
```
def calcTree(tree):
    # Fermiamo la ricorsione
    if isLeaf(tree): return tree.data
    # valutiamo gli operandi ricorsivamente
    leftOp = calcTree(tree.sx)
    rightOp = calcTree(tree.dx)
    if (tree.data == '+'):
        return leftOp + rightOp
    if (tree.data == '-'):
        return leftOp - rightOp
    if (tree.data == '*'):
        return leftOp * rightOp

print(calcTree(optree1))
print(calcTree(optree2))
print(calcTree(optree3))
```

Funziona, pero' e' un po' brutto. Non e' elegante la sequenza di 'if', e implica che per aggiungere o modificare le operazioni dobbiamo modificare la funzione. Cerchiamo di essere piu' funzionali.

Definiamo un dizionario che mappa da stringhe a funzioni, poi applichiamo la funzione appropriata. Il dizionario sara' un parametro della funzione

In [ ]:
```
# La funzione nuova
def calcTree(tree, ops):
    # Fermiamo la ricorsione
    if isLeaf(tree): return tree.data
    # valutiamo gli operandi ricorsivamente
    leftOp = calcTree(tree.sx, ops)
    rightOp = calcTree(tree.dx, ops)
    return ops[tree.data](leftOp, rightOp)

# definiamo il dizionario
ops = {'+': lambda x,y: x + y,
       '-': lambda x,y: x - y,
       '*': lambda x,y: x * y}

print(calcTree(optree1, ops))
print(calcTree(optree2, ops))
print(calcTree(optree3, ops))
```

Per aggiungere un'altra operazione basta aggiungerla al dizionario.

In [ ]:
```
ops['**'] = lambda x,y: x**y

optree4 = Tree('**',Tree(3),Tree(4))

calcTree(optree4, ops)
```

10. Esercizi su funzioni ricorsive

Caricate il file `10_Esercizi_funz_ric.py` e provate a fare gli esercizi proposti.