# Fondamenti di Informatica e Basi di Dati a.a. 2019/2020

DOCENTE: DOTT.SSA VALERIA FIONDA

basate sul materiale del Prof. **Mauro Gaspari**

# Introduzione alla programmazione con python

# PYTHON

## Funzioni

# Calcolabilità

- Anche se abbiamo visto solo un sotto insieme di python, questo sottoinsieme e' comunque un linguaggio di programmazione completo (con esso e' possibile calcolare qualsiasi funzione).
- Provare questo risultato non e' semplice (la dimostrazione si puo' trovare su testi di teoria della computabilita). Il risultato e' noto come Tesi di Turing (da Alan Turing).
- Per far vedere in modo sperimentale che questa tesi e' vera si da la realizzazione di alcune funzioni matematiche.

# Fibonacci

fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);

```
def fibonacci (n):
    if n == 0 or n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

# Fibonacci

- Cosa succede se si chiama il fattoriale con 1.5 come argomento?
- interprete....
- Come risolvere il problema?

```
def fibonacci (n):
    if type(n) != type(1):
        print "Factorial is only defined for integers."
        return -1
    if n == 0 or n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

# Osservazioni

- Ragionando si riesce a dimostrare che ora il programma di fibonacci termina sempre.
- Questo si puo fare grazie alle condizioni che sono state aggiunte dette guardie (= guardians).
- Le guardie permettono di dimostrare la correttezza del codice.

# PYTHON

## Iterazioni

# Il concetto di iterazione

- Spesso i computer sono utilizzati per eseguire in modo automatico task ripetitivi.
- Le operazioni ripetitive sono quelle che i computer riescono a fare meglio rispetto a noi (che spesso ci possiamo sbagliare).
- Tecnicamente la ripetizione di comandi si dice iterazione.

# Il comando **while**

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print "Blastoff!"
```

- Si puo' leggere il comando while in modo letterale: while significa “fino a quando” ed e' associato ad una condizione.
- Fino a quando la condizione vale si esegue il body del while.

# Semantica del **while**

1) Si valuta la condizione booleana.
2) Se la condizione e' falsa (0): si esce dal while e si continua con l'esecuzione del comando successivo.
3) Se la condizione e' vera (1): si eseguono tutti i comandi del body e alla fine si torna al passo 1.

Questo tipo di flusso di controllo si chiama anche ciclo (= loop), perchè con il passo 3) si torna indietro, all'inizio del comando.

# Osservazioni sul **while**

- Se la condizione e' falsa i comandi del body non vengono mai eseguiti.
- E' opportuno che nel body ci siano dei comandi che cambiano il valore di variabili che appaiono nella condizione.
- Altrimenti si ottengono dei loop infiniti.

# Il commando break

- Con il commando break è possible interrompere il ciclo anche nel caso in cui la condizione del while sia vera

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

# Il commando continue

- Con il commando continue è possible interrompere l'iterazione corrente e continuare con la successiva

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    continue
  i += 1
```

# Esempio

```
def sequence(n):
    while n != 1:
        print n
        if n%2 == 0:        # n is even
            n = n/2
        else:               # n is odd
            n = n*3+1
```

Questo programma termina sempre?

# Osservazioni

- Dato che a volte il numero cresce e a volte decresce non e' facile dimostrare in modo semplice che il programma termina sempre.
- E' possibile dimostrare la terminazione c'e' per qualche n. Ad esempio se n e' una potenza di 2.
- Ma e' possibile dimostrare che il programma termina per tutti i valori di n?
- Fino ad ora nessuno e' riuscito a dimostrare che questa affermazione vale e nemmeno che questa non vale.

# Non terminazione e decidibilita'

- Un problema e' decidibile quando si riesce sempre a dare una risposta.
- Ad esempio si puo' dimostrare e quindi decidere che il programma countdown visto prima termina sempre con gli interi positivi.
- Una proprieta' importante che vale nell'informatica e' la seguente: dato un qualsiasi programma non e' sempre possibile stabilire se questo termina per tutti i possibili input (**problema della non terminazione**)
- Quindi il problema della terminazione non e' decidibile.

# Esempio

```
i= 1
while i <= 6:
    print 2*i, ‘ ‘,
    i =i+1
print
```

# Incapsulamento e generalizzazione

- L'incapsulamento di un pezzo di codice si ottiene inserendo quel codice in una funzione.
- La generalizzazione si ottiene prendendo un pezzo di codice specifico e rendendolo piu' generale, adatto cioe' a risolvere altri problemi simili.
- Ad esempio si puo' generalizzare un pezzo di codice che stampa i multipli di 2 per fargli stampare i multipli di un numero qualsiasi.

# Esempio

```
def printMultiples(n):
    i =1
    while i <= 6:
        print n*i, ‘ ‘,
        i=i+1
    print
```

Questa funzione incapsula il codice visto in precedenza e lo generalizza per trattare multipli di qualsiasi numero.

# Osservazioni

- Per incapsulare un pezzo di codice quello che bisogna fare e' scrivere la prima linea con il nome della funzione e la lista dei parametri.
- Per generalizzare si rimpiazzano costanti con parametri.

# Esempio

```
i= 1
while i <= 6:
    printMultiples(i)
    i=i+1
```

E' possibile incapsulare e generalizzare ancora?

# Esempio

```
def printMultTable():
    i= 1
    while i <= 6:
        printMultiples(i)
        i=i+1
```

Si usa la stessa variabile nelle due funzioni ma questo non crea problemi: le variabili sono locali alle singole funzioni.

# Esempio

```
def printMultTable():
    i= 1
    while i <= 6:
        printMultiples(i)
        i=i+1
```

E' possibile realizzare una tabella della moltiplicazione di qualsiasi dimensione?

# Esempio

```
def printMultTable(high):
    i= 1
    while i <= high:
        printMultiples(i)
        i=i+1
```

E' possibile realizzare una tabella della moltiplicazione di qualsiasi dimensione?

# Generalizziamo ancora

- Mi serve una tabella NxN

```
def printMultiples(n, high):
    i=1
    while i <= high:
        print n*i, '\t',
        i=i+1
    print

def printMultTable(high):
    i= 1
    while i <= high:
        printMultiples(I,high)
        i=i+1
```

# Il comando **for**

- Un ciclo **for** viene utilizzato per iterare sugli elementi di una sequenza (ovvero un elenco, una insieme o una stringa).
- Con il ciclo for possiamo eseguire un set di istruzioni, una volta per ogni elemento in un elenco, tupla, set ecc.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print x
```

# La funzione **range()**

- Per eseguire un numero specificato di volte una sezione di codice possiamo usare la funzione range(),
- La funzione range() restituisce una sequenza di numeri, a partire da 0 per impostazione predefinita, e aumenta di 1 (per impostazione predefinita) e termina con un numero specificato.
- Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

```
for x in range(6):
  print x
```

# La funzione **range()**

- Per impostazione predefinita, la funzione range() ha 0 come valore iniziale, tuttavia è possibile specificare il valore iniziale aggiungendo un parametro:
- range (2, 6) specifica una sequenza di valori  da 2 a 6 (6 non incluso)

```
for x in range(2,6):
    print x
```

# La funzione **range()**

- Per impostazione predefinita, la funzione range() incrementa la sequenza di 1, tuttavia è possibile specificare il valore dell'incremento aggiungendo un terzo parametro

```
for x in range(2,30,3):
    print x
```

# Loop innestati

- Un loop innestato è un loop all'interno di un loop.
- Il "loop interno" verrà eseguito una volta per ogni iterazione del "loop esterno":

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print x, y
```