Laboratorio di matematica con Python

Release 0.6.1

Daniele Zambelli

01 February 2017

Indice

CAPITOLO 1

Introduzione a Python

1.1 Introduzione

1.1.1 Scopo di questo manuale

Che senso ha insegnare un linguaggio di programmazione nella scuola? Non certo quello di preparare futuri informatici, dato che:

- man mano che la tecnologia matura richiede sempre meno tecnici e tecnici sempre più specializzati;
- è molto difficile pensare di formare, con anni di anticipo, tecnici competenti per una tecnologia ancora così in evoluzione.

L'introduzione di ogni nuovo strumento tecnico-tecnologico modifica il modo di affrontare i problemi di una società e modifica anche i problemi che la società ritiene importante risolvere. Uno strumento così potente e multiforme come l'elaboratore elettronico ha un forte impatto nella cultura e può essere un forte stimolo per esplorare e capire il mondo che ci circonda.

Qui l'informatica, non tanto l'uso del computer, è vista come uno strumento ricco di stimoli per esplorare, imparare, crescere. Già, l'informatica... ma cos'è l'informatica?

''L'informatica è la scienza che studia come risolvere problemi usando il linguaggio" (Lucio Varagnolo anni '80).

Il linguaggio è il più potente (l'unico?) strumento di conoscenza umano. I linguaggi di programmazione, sono linguaggi formali, certo non paragonabili, in espressività, con il linguaggio naturale. Affrontare problemi, riprodurre e descrivere situazioni, confrontare l'effetto di diverse definizioni o algoritmi usando un linguaggio di programmazione, può comunque essere un potente stimolo ad una più profonda e personale conoscenza.

Il linguaggio di programmazione può dunque avere un senso nella scuola: non come fine dell'insegnamento, ma come mezzo per apprendere. Il linguaggio deve quindi risultare più "trasparente" possibile: non deve tediare e confondere, con una sintassi complicata, deve possedere delle strutture di alto livello in modo da permettere di affrontare problemi significativi molto presto, deve avere un rigoroso impianto logico in modo da risultare formativo.

Queste caratteristiche le ho trovate in Python che è anche un linguaggio multipiattaforma (funziona sotto Windows, Mac, Linux e molti altri sistemi operativi), ha una grande quantità di librerie che lo rendono adatto a risolvere problemi in ambiti molto diversi, ha una notevole documentazione, in parte tradotta anche in italiano da volontari.

Inoltre Python fa parte della grande famiglia di software libero, cioè: #. può essere usato senza restrizioni, #. può essere studiato e adattato alle proprie esigenze, #. può essere copiato liberamente e distribuito, #. può essere modificato, ma deve restare libero.

Insomma, attualmente, Python è un ottimo candidato come linguaggio di programmazione per la scuola secondaria.

1.1.2 Installazione

Per installare un qualunque software bisogna essere un po' pratici di computer. Conviene eventualmente farsi aiutare. L'autore non può essere ritenuto responsabile della perdita di informazioni conseguenti ad errori di installazione.

1. Installare Python
   (a) Python è un software libero ed è distribuito sotto una licenza compatibile con la licenza GPL quindi è possibile usarlo, copiarlo e distribuirlo senza restrizioni.
   (b) GNU/Linux: il programma deve essere installato dall'amministratore del sistema, lui sa come fare. Se si utilizza una distribuzione la cosa più semplice è installare Python dai repository della distribuzione stessa. Noi utilizzeremo per gli esempi l'ambiente di sviluppo IDLE (non è l'unica scelta possibile). In molte distribuzioni GNU/Linux IDLE è un pacchetto separato da Python, se lo vogliamo usare dobbiamo installarlo a parte.
   (c) Windows: scaricare la versione più aggiornata di Python da www.python.org quindi doppio clic sul pacchetto da installare.
2. Installare pygraph:
   (a) Copiare il file pygraph_x.xx" (nel mondo reale al posto di "x" ci saranno dei numeri che indicano la versione) all'interno di una propria cartella (il file può essere prelevato da Internet dalla pagina: https://www.verona.linux.it/index.php?title=Download) TODO
   (b) Scompattare il file "pygraph_x.xx" all'interno di una directory di lavoro.
   (c) La directory pygraph contiene le directory seguenti: * doc, la documentazione varia * examples, esempi d'uso * pygraph, le librerie del progetto:
      - pycart.py un piano cartesiano
      - pyplot.py grafico di funzioni nel piano, cartesiane e polari
      - pyturtle.py la grafica della tartaruga
      - pyig geometria interattiva
   (d) Spostare le directory doc e examples in una propria cartella facilmente raggiungibile (emailad esempio: .../mieidocumenti/python/pygraph)

(e) Spostare la directory pygraph e il file pigraph.pth all'interno di: .../pythonx.x/site-packages/ (osservazioni: 1: per fare questo in Linux bisogna avere i privilegi di amministratore; 2: a seconda della versione di Python installata, pythonx.x potrebbe essere python2.4, python2.5 o python3.0...)

3. Trovare documentazione su Python:

   (a) Documentazione in inglese: www.python.org

   (b) Documentazione in italiano: www.python.it

   (c) Un'ottima introduzione all'informatica usando questo linguaggio di programmazione è il testo: "How to think like a computer scientist: learning with Python" di Downey, Allen tradotto magnificamente in italiano (si trova su Internet partendo dai link precedenti).

1.1.3 Impostazione

In questo testo sono presentati vari strumenti che il linguaggio mette a disposizione del programmatore. Per ogni argomento vengono date succinte informazioni, viene indicata la sintassi delle nuove istruzioni, vengono proposti degli esempi e degli esercizi. Non mi sono posto l'obiettivo di trattare in modo completo le caratteristiche del linguaggio Python, ma scrivere solo dei cenni che ne permettano l'uso. Su ogni argomento si possono trovare informazioni più complete in internet e in particolare nei vari testi (anche tradotti in italiano) presenti nel sito italiano di Python: www.python.it.

1.1.4 Feedback

Spero che qualcuno trovi interessante questo lavoro e lo usi per imparare o per insegnare l'informatica e la geometria con Python. Utilizzandolo, senz'altro verranno alla luce molti errori, difetti o carenze sia nel software sia nella documentazione; chiedo a chi troverà qualche motivo di interesse in questo lavoro di inviarmi:

1. Commenti,
2. Critiche,
3. Suggerimenti,
4. ...e magari anche altri materiali di lavoro per accrescere la documentazione

Ringrazio fin d'ora chiunque perda del tempo a leggere questo testo, lo utilizzi nell'insegnamento o si prenda la briga di mandarmi qualche riscontro.

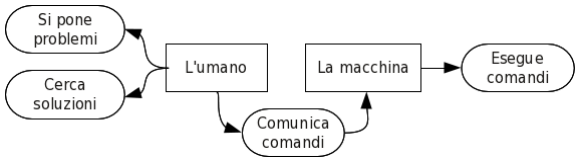
1.1.5 Licenza

Daniele Zambelli: daniele.zambelli at inwind.it

1.2 I primi comandi

Cos'è un linguaggio di programmazione, la prima istruzione, come scrivere ed eseguire un semplice programma.

L'informatica è la scienza che usa il linguaggio per risolvere i problemi. L'uomo trova la soluzione del problema, poi comunica alla macchina quali passi deve eseguire per trovare il risultato desiderato. La macchina esegue i comandi ricevuti.



Per comunicare alla macchina usiamo un dispositivo che è la tastiera. Su questa possiamo scrivere i comandi che vogliamo la macchina esegua. Ad esempio: "preparami un paninazzo con la soppressa!"

No, questo comando non verrà eseguito dalla macchina. Possiamo chiederle di eseguire solo azioni per cui è stata realizzata e dobbiamo chiedere di eseguire quelle azioni usando un linguaggio che lei possa comprendere.

Purtroppo il linguaggio che fa funzionare i computer è piuttosto oscuro:

```
1000110101000110
0011111010101110
1100010101101111
...
```

e è detto linguaggio macchina.

Dopo qualche anno di programmazione con parole fatte di "zeri" e "uni", gli informatici hanno inventato dei programmi capaci di tradurre i comandi scritti in un linguaggio comprensibile dagli umani, nel linguaggio macchina comprensibile dagli elaboratori elettronici.

Python è un interprete di linguaggio che traduce comandi che possiamo capire noi, in comandi che la macchina capisce ed esegue. Ovviamente non possiamo dare dei comandi come ci pare e piace, dobbiamo seguire le regole di un linguaggio, obbedire ad una precisa grammatica ed usare solo le parole che Python capisce.

Come primo comando diciamo a Python di scrivere qualcosa sullo schermo: il comando per dire questo è "stampa" forse perché una volta i computer non avevano lo schermo, ma solo una tastiera e una stampante. Il comando però va scritto in inglese (chissà perché Python non conosce l'italiano?)

```
print
```

Ma non basta dire "stampa", bisogna anche dire che cosa deve essere mostrato, il primo comando che produce qualcosa di significativo è dunque:

```
print "Ciao gente!"
```

`print` è un'istruzione di Python e la parte di comando compreso tra le virgolette è una stringa di caratteri.

Ma come facciamo a dire a Python di leggere ed eseguire questo comando? Chi ha implementato il linguaggio ha anche realizzato e messo a disposizione un ambiente per lo sviluppo dei programmi: IDLE.

IDLE, che è un programma scritto in Python, permette di scrivere comandi e osservare immediatamente il loro effetto, è uno strumento molto comodo. Scoviamo dove il programma di installazione ha messo IDLE ed eseguiamolo (per Windows: `Start-Programmi-Python2.7-IDLE`, per Linux, se non si trova nel menu delle applicazioni, si può scrivere `idle` oppure `idle-python2.7` in una finestra di terminale). Una volta avviato l'ambiente di sviluppo, sotto ad alcuni messaggi che riportano la versione di Python (controllare se è "2.x" o "3.x"!) in uso e di IDLE stesso, appaiono tre simboli i maggiore e il cursore lampeggiante subito dopo:

```
>>>
```

Il significato di questi simboli è: "sono pronto, cosa vuoi che esegua?". Bene, qui possiamo dare il nostro comando. Questo comando dipende dalla versione di Python in uso. Se usiamo Python 2.x il comando è:

```
>>> print "Ciao gente!"
```

Se usiamo Python 3.x il comando è:

```
>>> print("Ciao gente!")
```

Posso anche far stampare più stringhe con uno stesso comando, in Python 2.x:

```
>>> print "Ciao", "gente!"
```

Se usiamo Python 3.x:

```
>>> print("Ciao", "gente!")
```

In questo caso le stringhe devono essere separate da virgole. Python inserisce automaticamente uno spazio tra una stringa e l'altra. Posso anche inserire nella stringa dei caratteri speciali:

```
>>> print "Ciao\n\n\n\tgente!"
```

o:

```
>>> print("Ciao\n\n\n\tgente!")
```

Il significato della stringa "\n" è: "qui va a capo"... scopri tu qual è il significato di "\t".

Riassumendo

- Un linguaggio di programmazione è una lingua artificiale che può essere tradotta dal calcolatore elettronico in una sequenza di comandi eseguibili.
- Un programma è una sequenza di istruzioni come.
- Nel passaggio da Python 2.x a Python 3.x è stato modificata l'istruzione `print` in Python 2.x `print` è un comando: `print <stringa>`, in Python 3.x è una funzione: `print(<stringa>)`
- `<stringa>` è una sequenza di caratteri racchiusa tra apici singoli o doppi ad es.:

```
"Ciao gente\n"
```

- La sequenza di caratteri "\n" non viene stampata ma indica all'interprete che deve andare a capo (new line).
- La sequenza di caratteri "\t" significa ...

Prova tu

1. Scrivi comandi che stampino diverse frasi.
2. **Scrivi un comando che stampi diverse frasi una sotto l'altra, ad es.:** Nel mezzo del cammin di nostra vita mi ritrovai per una selva oscura che la diritta via era smarrita. (Dante Alighieri)
3. Descrivi cosa succede se in un comando print aggiungo una virgola dopo l'ultima stringa.

1.3 Python come "calcolatore"

Come far fare calcoli a Python.

Bene, abbiamo imparato a far scrivere qualcosa a Python, ma la faccenda è un po' stupida dato che siamo noi a dire esattamente cosa ci deve mostrare. Vediamo di chiedergli qualcosa di più... Cosa c'è di meglio di chiedere a un "calcolatore" se non di far calcoli? Mettiamolo alla prova. Possiamo scrivere istruzioni per stampare il risultato delle operazioni aritmetiche:

```
>>> print(5+9)
14
```

Alcune osservazioni:

1. L'espressione "5+9" può anche contenere spazi: "5 + 9" e viene comunque interpretata ed eseguita correttamente.
2. All'interno di IDLE si potrebbe anche omettere il comando print, l'ambiente IDLE aggiunge automaticamente un'istruzione per la visualizzazione del risultato di un'espressione:

```
>>> 5+9
14
```

   ma è preferibile evitare comandi impliciti: "Non basta che tu calcoli 5+9, devi anche stampare il risultato!".

3. Nota: l'ambiente IDLE (ma cosa vuol dire idle in inglese?) permette di riprendere e modificare un'istruzione precedente senza riscriverla, basta portare il cursore sull'espressione, premere <Invio> e IDLE ce la ricopia su una nuova riga, pronta per essere modificata e rieseguita.

Prova tu scrivendo espressioni con più operazioni, espressioni con parentesi controllando che Python faccia i calcoli giusti!

Se hai fatto diverse prove e controllato con attenzione i risultati ti sarai accorto che:

1. Python commette degli errori,
2. Python sa eseguire poche operazioni.

Il primo problema sorge quando tentiamo di eseguire una divisione:

```
>>> print 33/7
4
```

Ma come è possibile che un linguaggio di programmazione serio faccia questi errori? I programmatori che hanno progettato Python hanno pensato che il risultato di una divisione tra interi debba essere un numero intero. Se vogliamo vedere i decimali del risultato, almeno uno degli operandi deve essere un numero con la virgola (la virgola nei linguaggi di programmazione è il punto!):

```
>>> print 33./7
4.71428571429
```

Quindi Python tratta in modo diverso "33" da "33.": il primo è un oggetto `int` (intero), il secondo un oggetto `float` (in virgola mobile). È importante, quando si affronta un problema, prestare ben attenzione se possiamo trattarlo usando numeri interi o in virgola mobile per evitare errori difficili da scovare.

Chi sta usando Python 3.x non avrà compreso i paragrafi precedenti infatti in Python 3.x è stato cambiato il comportamento della divisione, il risultato è sempre un numero razionale (`float`) anche quando gli operandi sono numeri interi.

Nel resto di questo manuale utilizzeremo Python 3.x. Ma chi ha installata sulla propria macchina una versione 2.x può dire a Python di anticipare il comportamento della 3.x importando la divisione e la funzione `print` dal modulo `__future__` con il seguente comando:

```
>>> from __future__ import division, print_function
```

Dopo questa istruzione anche Python 2.x darà il seguente risultato:

```
>>> print(33/7)
4.71428571429
```

Riassumendo

- Possiamo usare Python come una calcolatrice. Facendogli calcolare il risultato di espressioni complesse quanto vogliamo.
- Si può far riscrivere a IDLE un'espressione, modificarla e rieseguirla.

- Possiamo chiedere a Python 2.x di copmortarsi come Python 3.x per quanto riguarda il comando `print` e la divisione eseguendo l'istruzione:

```
>>> from __future__ import division, print_function
```

Prova tu

1. Da un libro di matematica prendi qualche espressione con numeri interi e falla eseguire a Python controllando il risultato.
2. Ora prova a far eseguire espressioni con le frazioni.
3. Fa qualche prova per vedere quali parentesi vengono accettate nelle espressioni matematiche.
4. Scrivi le espressioni che calcolano il perimetro e l'area dei quadrilateri che conosci.

1.4 La libreria `math`

Come sfruttare le capacità matematiche di Python.

Nell'eseguire espressioni matematiche di sicuro ci si imbatte in un altro problema (oltre a quello della divisione tra interi). Se tentiamo di eseguire una radice quadrata ("sqrt" in inglese), Python ci risponde con un messaggio di errore:

```
>>> sqrt(16)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
    sqrt(16)
NameError: name 'sqrt' is not defined
```

È l'ultima riga quella che ci interessa in questo momento: Python non conosce la parola "sqrt", non sa fare le radici quadrate. Nei linguaggi moderni, la tendenza è quella di avere un nucleo di linguaggio piuttosto ristretto, ampliato poi da librerie. Python possiede una grande quantità di librerie, una di queste è "math" che contiene le funzioni matematiche. Per poterle usare dobbiamo caricarla e poi chiamare la funzione desiderata specificando il nome della libreria in cui si trova:

```
>>> import math
>>> print math.sqrt(16)
4.0
```

Da notare che il risultato della funzione math.sqrt() è un numero float (razionale o in virgola mobile) anche se la parte decimale è nulla.

Le Librerie contengono, in generale un gran numero di funzioni, per avere informazioni sulla libreria possiamo usare il comando "help":

```
>>> help(math)
```

In questo modo vengono elencati tutti gli oggetti forniti dalla libreria `math`. Ci sono le funzioni goniometriche e iperboliche, la conversione da radianti in gradi e viceversa, la parte intera di

un numero, le funzioni esponenziale e logaritmica, il valore assoluto, il fattoriale, la potenza di un numero e la radice quadrata, la lunghezza dell'ipotenusa dati i cateti e le due costanti `e` e `pi`.

Ad esempio se volessimo stamparci una tabella dei valori di seno, coseno e tangente per gli angoli che vanno da 0 a 360 gradi con intervalli di 15 gradi potremmo andare per gradi... Iniziamo a farci stampare il valore degli angoli:

```
>>> from __future__ import print_function
>>> for alpha in range(0, 360, 15):
        print(alpha)


0
15
30
...
```

Osserviamo che `range` è un *iteratore* che fornisce, uno alla volta tutti i numeri da un minimo opzionale (in questo caso 0), a un massimo sempre escluso (in questo caso 360), con un certo intervallo opzionale (in questo caso 15).

L'istruzione `for` permette di ripetere un blocco di istruzioni per ogni valore di una sequenza (in questo caso fornita da `range`). Verrà trattato in un prossimo capitolo.

Aggiungiamo ora il calcolo del seno:

```
>>> for alpha in range(0, 360, 15):
        print(alpha, math.sin(alpha))


0 0.0
15 0.650287840157
30 -0.988031624093
...
```

Ma qui si evidenziano due problemi: primo vorrei i numeri incolonnati, secondo sbaglia i calcoli! Per il primo problema aggiungiamo alla funzione `print` l'argomento relativo al separatore: `sep='\t'` in questo modo i diversi valori vengono separati da un tabulatore. Per il secondo problema, riguardiamo la documentazione (`help(math)`) la funzione `sin` vuole come argomento l'angolo espresso in radianti. Nessun problema, `math` ha giusto una funzione per la conversione da gradi a radianti. Magari arrotondiamo anche i valori a 4 cifre decimali in modo da renderli un po' più leggibili:

```
>>> for alpha in range(0, 360, 15):
        print(alpha, round(math.sin(math.radians(alpha)), 4), sep='\t')


0       0.0
15      0.2588
...
```

Riassumendo

- La libreria math permette di estendere le funzionalità matematiche di Python per usarla la si deve prima caricare con il comando `import`.
- La funzione help() può darci interessanti informazioni su un oggetto Python.
- Possiamo chiedere a Python 2.x di comportarsi come Python 3.x per quanto riguarda il comando `print` e la divisione eseguendo l'istruzione:

```
>>> from __future__ import division, print_function
```

- L'iteratore `range` fornisce numeri interi, la sintassi è:

```
range([min, ] max [, step])
```

- L'istruzione `for` permette di ripetere un blocco di istruzioni facendo scorrere una variabile in una sequenza, la sintassi è:

```
for <variabile> in sequenza:
    <istruzioni>
```

Prova tu

1. Fa calcolare la lunghezza dell'ipotenusa di un triangolo rettangolo con i cateti lunghi 13 e 18.
2. Modifica l'espressione precedente in modo che calcoli le ipotenuse di triangoli rettangoli con cateti lunghi: 643 e 786, 32332 e 57543, 775472 e 547545.
3. Scrivi le espressioni che calcolano il perimetro e l'area del triangolo equilatero.
4. Completa la tabella delle funzioni goniometriche con la colonna del coseno e della tangente.

1.5 Dati: nomi e oggetti

Come memorizzare dei dati e richiamarli quando serve.

Abbiamo visto il modo di far eseguire delle istruzioni al calcolatore, ma ciò servirebbe ben a poco se non potessimo anche inserire dei dati nei nostri programmi. In Python i dati sono degli *oggetti* che l'interprete memorizza da qualche parte (e si premura di cancellare, per liberare memoria, quando non servono più). Alcuni di questi oggetti sono *immutabili*, il numero 5 sarà sempre 5, non può diventare 7. Altri sono *mutabili*: la *lista* che contiene il numero 5 può diventare una *lista* che contiene il numero 7. In un programma, noi possiamo fare riferimento a questi oggetti usando dei nomi detti *identificatori*. Il meccanismo che permette di collegare un *identificatori* ad un *oggetto* si chiama assegnazione e, in Python, viene indicata dal simbolo: "=".

```
>>> data = "13-10-2001"
```

`data` è l'identificatore = è l'operatore di assegnazione e `13-10-2001` è un oggetto stringa. La riga precedente significa:

```
collega all'identificatore data l'oggetto "13-10-2001"
```

Per l'identificatore ci sono alcune restrizioni: non può iniziare con una cifra e non può contenere lettere accentate o alcuni altri caratteri particolari. Così per memorizzare il nome e cognome di una persona, la sua età e il suo indirizzo posso scrivere:

```
>>> nome = "Mario Rossi"
>>> eta = 45
>>> indirizzo = "via Verdi, 4"
```

Python sembra non reagire ai comandi precedenti, il fatto che non stampi messaggi di errore è comunque confortante. Python ha scritto le due stringhe e il numero intero in una zona di memoria e ha associato a questi oggetti tre nomi. In questo modo abbiamo legato all'identificatore `nome` la stringa "Mario Rossi", all'identificatore `eta` il numero 45 e all'identificatore `indirizzo` la stringa "via Verdi, 4". Dati questi comandi possiamo usare l'identificatore al posto dell'oggetto:

```
>>> print(nome, eta, indirizzo)
Mario Rossi 45 via Verdi, 4
```

Ora se volessimo stampare una scheda in formato "burocratichese", con i dati di Mario Rossi, usando un solo comando, potremmo scrivere:

```
>>> print "Sig.", nome, "\nabitante in", indirizzo, "\ndi anni", eta
Sig. Mario Rossi
abitante in via Verdi, 4
di anni 45
```

Riprendiamo le espressioni matematiche. Se voglio scrivere un'espressione e il suo risultato, posso dare il comando:

```
>>> print "2 + 3 =", 2+3
2 + 3 = 5
```

Ma se devo cambiare gli operandi, devo intervenire in più punti dell'istruzione precedente e questo non è carino. Posso usare invece le variabili:

```
>>> a=7
>>> b=5
>>> print(a, "+", b, "=", a+b)
7 + 5 = 12
```

In questo modo, cambiando il valore delle variabili posso usare sempre la stessa istruzione per visualizzare un gran numero di espressioni diverse:

```
>>> a = 73584784689
>>> b = 5647468959
>>> print(a, "+", b, "=", a+b)
```

A prima vista, può sembrare che non vi sia un gran vantaggio in questo modo di procedere. Comandare le tre linee precedenti sembra più lungo, complicato e noioso che utilizzare direttamente i valori, come sarebbe per esempio:

```
>>> print(73584784689, "+", 5647468959, "=", 73584784689+5647468959)
```

È un modo barbaro, ma, in fondo, dà lo stesso risultato. Tra i due metodi c'è una importante differenza di impostazione. Con un paio di righe di comando in più si ottiene la separazione dei dati dal loro uso. Una volta assegnate le variabili `a`, `b`, esse sono riutilizzabili per altri calcoli. Prova:

```
>>> a = 100
>>> b = -2
>>> print("somma =",a+b, "prodotto =", a*b, "potenza =", a**b)
```

A questo punto, la linea di comando può essere riusata con altri dati. Per esempio:

```
>>> a = 27.5
>>> b = -34.21
```

e poi si ricopia la linea (con un clic e `<Invio>`, come già visto) e la si conferma con `<Invio>`.

Riassumendo

- Una variabile è un nome associato ad un oggetto.
- L'uso di variabili rende più flessibili i programmi.
- Ad una variabile si può associare un valore con l'istruzione di assegnazione:

```
    <nome> = <oggetto>
```

- Gli oggetti che abbiamo usato finora sono: numeri interi e numeri in virgola mobile.

Prova tu

1. Usando le variabili scrivi l'espressione per calcolare l'area dei quadrilateri notevoli.
2. Dopo aver messo in una variabile di nome "n" un numero intero scrivi l'istruzione che ne stampi la sua tabellina. (>>> print 1*n, 2*n, ...). Stampa la stessa tabellina in colonna.
3. Un gioco dice così: "Prendi un numero intero. Raddoppialo. Aggiungi 10. Poi dimezza il risultato. Togli il numero che hai pensato. Scommetti che risulta 5?" Scrivi l'espressione e prova con diversi valori della variabile. Secondo te perché è proprio vero che risulta 5?

1.6 Stringhe

Come usare dati che sono sequenze di altri dati.

Nel capitolo precedente abbiamo creato una variabile `data` associata all'oggetto stringa: `"13-10-2001"`. Una stringa è una sequenza di caratteri. È possibile estrarre ogni elemento di una sequenza usando gli indici. Dobbiamo tenere presente che in Python le sequenze iniziano con l'indice `0`, quindi se vogliamo ottenere il primo carattere di una stringa dobbiamo estrarre l'elemento di indice `0`:

```
>>> data = "13-10-12"
>>> print(data[0])
1
```

È anche possibile usare un indice che conta partendo dalla fine della sequenza si ottiene questo effetto usando come indice un numero negativo:

```
>>> print(data[-1])
2
```

Per mezzo dell'indicizzazione si può ottenere un elemento della sequenza, ma uno strumento simile permette anche di ottenere una sotto sequenza. Ad esempio per ottenere i primi due elementi posso scrivere:

```
>>> print(data[0:2])
13
```

Da notare che l'istruzione precedente restituisce 2 elementi quello di posizione `0` e quello di posizione `1` (cioè esattamente 2-0 elementi). Questa operazione si chiama "affettamento" ("slicing") il risultato è sempre primo elemento compreso e ultimo escluso.

Se volessi estrarre la parte di stringa relativa al mese dovrei scrivere:

```
>>> print(data[3:5])
10
```

Da notare che gli elementi estratti sono quello di indice 3 e quello di indice 4,

In una fetta di una sequenza il primo e l'ultimo elemento possono essere esclusi, così l'istruzione precedente è equivalente a:

```
>>> print(data[:2])
13
```

Se voglio gli ultimi 2 caratteri posso scrivere:

```
>>> print(data[-2:])
12
```

Che significa: "dal penultimo compreso fino alla fine della sequenza". A volte può tornare utile ottenere tutti gli elementi di una sequenza in questo caso l'istruzione è:

```
>>> print(data[:])
13-10-12
```

A volte servono degli elementi che non si trovano di seguito ma che sono disposti ad intervalli regolari. Ad esempio se dalla solita data dovessi estrarre i due trattini, dovrei estrarre i caratteri di posizione 3 e di posizione 6 cioè un carattere ogni 3. L'istruzione che fa questo è:

```
>>> print(data[::3])
--
```

In Python si possono scrivere espressioni con le stringhe: una stringa più un'altra stringa dà come risultato la concatenazione dei due operandi:

```
>>> s0 = "cocco"
>>> s1 = "drillo"
>>> s2 = "bello"
>>> print(s0+s1)
```

```
coccodrillo
>>> print(s0+s2)
coccobello
```

Una stringa può anche essere moltiplicata per un numero intero, che risultato darà?

Esistono altri oggetti si tipo sequenza oltre alle stringhe: `tuple`, `liste`. Questi oggetti sono sequenze di altri oggetti qualunque, cioè possono contenere numeri, stringhe, ma anche `tuple` o `liste` o qualunque altro oggetto Python. Questi tipi e altri tipi di oggetti Python verranno trattati più avanti. Ma anticipiamo qui alcuni metodi dell'oggetto stringa che permette di trasformare una stringa in una lista di parole e viceversa di ricostruire una stringa a partire dalla lista:

```
>>> stringa = 'sopra la panca la capra campa'
>>> stringa.split()
['sopra', 'la', 'panca', 'la', 'capra', 'campa']
>>> lista = stringa.split()
>>> lista
['sopra', 'la', 'panca', 'la', 'capra', 'campa']
>>> ' '.join(lista)
'sopra la panca la capra campa'
```

Riassumendo

- Una stringa è una successione ordinata di caratteri.
- Per estrarre gli elementi di una successione si usa l'indicizzazione:

```
    <lista>[<indice>].
```

- Il primo elemento di una successione ha indice 0, l'ultimo ha indice -1.
- Si può anche estrarre una fetta di una successione (slicing):

```
    <successione>[inizio:fine]
```

  che restituisce gli elementi della successione da inizio compreso a fine escluso.

Prova tu

1. Data la seguente stringa: parola = "deambulare" scrivi in due modi diversi l'istruzione che stampi il suo quinto carattere.
2. Data la stringa precedente scrivi le istruzioni per stampare le sotto stringhe: "dea", "re", "bula", "emuae",
3. Costruisci la stringa che contenga dieci volte la parola "Ciao".
4. Costruisci la stringa che contenga dieci righe ciascuna contenente dieci "Ciao".

1.7 Convertire, formattare

Come far stampare a Python quello che vogliamo noi.

Abbiamo visto che esistono diversi tipi di oggetti: `int`, `float`, `string`, `list`, ... È indifferente usare un tipo o un altro? Proviamo a chiederlo direttamente a Python, costruiamo 3 oggetti che contengono il numero 7:

```
>>> i = 7
>>> f = 7.
>>> s = "7"
```

e confrontiamoli tra di loro:

```
>>> i == f
True
>>> i == s
False
>>> f == s
False
```

Alcune osservazioni:

1. Come vedi il simbolo = viene usato in due modi. Nel solito modo funge da operatore di assegnazione. Invece, se ripetuto due volte (così: ==) serve a confrontare due valori.
2. Se i due valori sono uguali, il confronto darà come risultato `True`, altrimenti darà come risultato `False`.
3. `True` e `False` sono particolari oggetti di Python, ed indicano, ovviamente, ciò che tutti noi intendiamo per vero e falso.

Evidentemente i contenuti delle variabili precedenti non sono tutti uguali! E il primo caso? Al suo interno Python tratta in modo diverso numeri interi e numeri razionali o in vircola mobile (float), ma quando serve converte gli interi in razionali.

Python mette a disposizione delle funzioni che cercano di trasformare un oggetto di un tipo in un oggetto di un altro tipo. È possibile trasformare un numero intero in un numero numero in virgola mobile e viceversa (ovviamente perdendo i decimali), una stringa che contiene un numero in un numero corrispondente e viceversa, un numero in una stringa. Se nella shell omettiamo l'istruzione `print`, `IDLE` mostra il dato con alcune informazioni in più.

```
>>> i
7
>>> f
7.0
>>> s
'7'
>>> int(f)
7
>>> str(i)
'7'
>>> float(s)
7.0
```

Nel passaggio da Python 2 a Python 3 il comportamento della divisione è cambiato: in Python 3 la divisione dà sempre come risultato un numero decimale (in questa versione anche l'istruzione `print` si comporta in modo differente). Quanto segue riguarda il comportamento di Python 2.

Supponiamo di avere nella variabile `a` il numero 15 e nella variabile `b` il numero 4 e di voler ottenere la stampa di questa stringa: "15/4=3.75". Creiamo le nostre variabili e usiamo i metodi che già conosciamo:

```
>>> a = 15
>>> b = 4
>>> print a, "/", b, "=", a/b
15 / 4 = 3
```

La divisione dà solo la parte intera del quoziente, ma se trasformiamo una variabile intera in virgola mobile otteniamo il risultato corretto:

```
>>> print a, "/", b, "=", float(a)/b
15 / 4 = 3.75
```

I calcoli ora sono esatti, ma non mi piacciono gli spazi inseriti. Per eliminarli, posso evitare le virgole, trasformare in stringhe ogni numero e concatenare le stringhe:

```
>>> print str(a) + "/" + str(b) + "=" + str(float(a)/b)
15/4=3.75
```

Ottengo esattamente quello che volevo, ma l'istruzione è penosa da scrivere... Python mette a disposizione un altro metodo per ottenere lo stesso risultato: la formattazione di stringhe. Scrivo la stringa come voglio ottenerla, ma al posto delle variabili metto dei segnaposto: "{}", a questo punto viene chiamato il metodo `format` passandogli gli argomenti che sostituiranno i segnaposti.

```
>>> print("{}/{}={}".format(a, b, a/b))
15/4=3.75
```

Riassumendo

- È possibile entro certi limiti trasformare un oggetto in un altro.
- Questo permette di trattare interi come float, stringhe come interi, ...
- È possibile costruire stringhe con il metodo `format`:

```
    <stringa con segnaposti>.format(oggetto1, oggetto2, ...)
```

  usando come segnaposti "{}".

Prova tu

1. Prova le seguenti linee di comando e prendi nota di ciò che fa Python.

```
>>> int(10)
>>> int("10")
>>> int(10.3)
>>> float(10)
```

```
>>> float("10")
>>> float(10.3)
>>> str(10)
>>> str("10")
```

```
>>> print(str(1.3)+'1')
>>> print(str(10)*5)
>>> print(float(10)*5)
>>> print(float(5)/2)
```

2. Riprendi gli esercizi 3, 5, 6 del capitolo riguardante le espressioni e costruisci delle stringhe con la risposta completa al problema usando la conversione di tipo con la concatena-

zione e poi usando la formattazione di stringhe (ad es. “L’ipotenusa di un triangolo con i cateti lunghi 3 e 4 è lunga 5”).

3. Metti il valore 5 nella variabile base e il valore 3 nella variabile esponente, poi scrivi il comando per ottenere la stringa: “5^3=125” usando le funzioni di conversione e la formattazione di stringa.
4. Metti in una variabile un numero intero, poi scrivi l’istruzione che produce la stringa formata dal numero della variabile, dal suo quadrato e dal suo cubo, separati da un tabulatore.

1.8 Ripetere istruzioni

Come far stare mille comandi in una riga di codice.

I computer attuali eseguono milioni o miliardi di istruzioni al secondo, ma se devo scrivere esplicitamente ogni istruzione che deve eseguire, impiego secoli e secoli per scrivere un programma. Se voglio che un programma stampi 3 volte di seguito una frase posso scrivere:

```
>>> print "ciao a tutti"; print "ciao a tutti"; print "ciao a tutti";
ciao a tutti
ciao a tutti
ciao a tutti
```

oppure posso usare un’istruzione che produce un ciclo:

```
>>> for cont in range(3):
                print "ciao a tutti"


ciao a tutti
ciao a tutti
ciao a tutti
```

Alcune osservazioni:

1. L’istruzione `for ...` non è molto più sintetica dei 3 comandi messi di seguito. Ma se dovessi stampare 1000 volte la frase, col primo metodo la faccenda diventerebbe complicata e noiosa, mentre con l’istruzione for ... mi basterebbe cambiare un numero:

```
    for cont in range(1000): ...
```

2. L’istruzione va capita, partiamo dal fondo:
   (a) `range(1000)` fornisce, una lista che contiene gli interi da 0 a 999 (1000 numeri),
   (b) uno alla volta, i numeri contenuti nella lista prodotta da range, viene messo nella variabile `cont` ...
   (c) e per ogni valore di `cont` viene eseguito il blocco di codice che segue il simbolo “:”
3. La sintassi è:

```
    for <variabile> in range(<numero>):
        <istruzioni>
```

4. L'istruzione `for` ripete tutte le istruzioni che la seguono e che sono indentate, cioè rientrate di alcuni spazi rispetto all'istruzione stessa.
5. In Python le righe di codice consecutive che iniziano alla stessa colonna, formano un blocco di codice.
6. Se avviamo un ciclo troppo lungo e vogliamo interromperlo possiamo premere il tasto: <Ctrl-c>.
7. Il tipo di struttura informatica prodotta dal comando for si chiama iterazione.

Proviamo ad esplorare il significato della variabile che segue l'istruzione for, a cosa può servite? Iniziamo a provare il comando:

```
>>> for cont in range(5):
                print cont


0
1
2
3
4
```

Possiamo farci stampare una tabella di quadrati e cubi:

```
>>> for num in range(11):
        print "%s  %s  %s" % (num, num*num, num*num*num)
```

ma i numeri sono allineati male. Prova questa variante:

```
>>> for num in range(5):
        print "%5s  %5s  %5s" % (num, num*num, num*num*num)
```

Che significato ha il numero inserito nel segnaposto? E per vedere i reciproci dei numeri precedenti? Ovviamente (?!) non posso partire da 0, ma da 1. Devo anche stare attento alle espressioni: 1/i*i non è lo stesso di 1/(i*i). Poiché sia 1 che i sono valori interi, Python tronca il risultato eliminando i decimali. Un modo per risolvere i problemi precedenti è:

```
>>> for n in range(1, 5):
        print "%10s  %10s  %10s" %(1./n, 1./(n*n), 1./(n*n*n))
```

Riassumendo

- Possiamo far ripetere a Python un blocco di codice usando l'istruzione `for`.
- Un blocco di codice si riconosce perché è formato da istruzioni consecutive che iniziano in modo allineato, sulla stessa colonna.
- La sintassi dell'istruzione for è:

```
for <variabile> in range(<numero>):
    <bloco di istruzioni>
```

- La variabile del ciclo contiene, ad ogni ciclo, un numero intero successivo da 0 a <numero> escluso.
- `range` può avere due parametri in questo caso la variabile va dal primo all'ultimo escluso.

Prova tu

1. La formattazione dell'ultimo esempio, fa un po' schifo... prova con quest'altra stringa: "%10.5f %10.5f %10.5f"
2. Quali altri caratteri di formattazione sono riconosciuti da Python?
3. Scrivi un ciclo che stampi per ogni numero da 0 a 20: il numero stesso, il suo doppio il suo triplo e il suo quadruplo, in modo che risultino allineati.
4. Metti in una variabile, "n" un numero. Scrivi il ciclo che stampi i suoi primi 15 multipli.
5. Scrivi un ciclo che stampi i multipli di un numero senza andare a capo.
6. Scrivi un ciclo che stampi i multipli di 7 ma solo dal 35° multiplo al 43°.
7. Un ciclo può contenere altri cicli annidati... Scrivi il comando che stampi la tabellina pitagorica.

1.9 Il primo programma

Come scrivere il nostro primo programma, un programma che non fa niente, un programma che fa qualcosa.

Finalmente abbiamo gli strumenti per scrivere qualcosa di significativo, qualcosa di abbastanza complicato da valer la pena di conservare, un programma. Aperto `IDLE`, attraverso il menu `File - New Window` creiamo un nuovo file vuoto. Prima ancora di incominciare a scriverci qualcosa lo salviamo, nella nostra cartella, con il nome "primo.py". È importante l'estensione ".py" che avvisa il sistema operativo, e `IDLE` stesso, che questo è un programma Python. Abbiamo scritto il nostro primo programma! Possiamo anche eseguirlo in uno di questi due modi:

1. Menu: Run - Run Module
2. tasto: <F5>

Fa un po' poco, ma in compenso, di sicuro, non contiene errori!!! Forse è il caso di cominciare a riempirlo.

Per prima cosa aggiungiamo delle istruzioni che non fanno niente, dei commenti. Ogni linguaggio permette di scrivere dei commenti. E i commenti sono molto importanti per rendere leggibile il codice e per renderlo comprensibile ad altri o anche a sé stessi a distanza di tempo. In Python ci sono più modi per farlo. Sono commenti:

1. le righe che iniziano con il carattere cancelletto: "#"

2. le porzioni di testo delimitati da tre apici singoli o doppi: “”“”“” o “‘‘‘”

Ogni programma deve avere all'inizio dei commenti che forniscono alcune informazioni come minimo: data, autore, titolo. Il commento iniziale potrebbe essere:

```
# 27-08-11
# il mio primo programma
# Mario Rossi
```

Prima di scrivere un programma, bisogna avere un'idea abbastanza precisa di cosa deve fare questo programma. Nel nostro caso iniziamo a creare un programma che stampi a video le tavole numeriche, quelle con i quadrati, i cubi,le radici quadrate e radici cubiche. Per prima cosa modifichiamo il titolo del programma. Poi (magari andando a riguardare le informazioni sui cicli) scriviamo le istruzioni per stampare semplicemente i numeri da 0 a 100:

```
for num in range(101):
    print "%s" % num
```

Alcune osservazioni:

1. Mentre nella `shell` di `IDLE` una volta scritto un comando, quando si preme il tasto `<Invio>`, magri 2 volte, il comando viene eseguito, qui il tasto `<Invio>` ha solo il significato di andare a capo.
2. In un programma, noi scriviamo le istruzioni, per eseguirle dobbiamo dirlo esplicitamente a Python, lo si può fare in uno dei due modi visti sopra, il più svelto è premere `<F5>`.
3. Prima di eseguire un programma, Python lo salva sul disco.
4. Il risultato del programma (l'output) lo vediamo nella finestra di IDLE. Conviene quindi disporre le due finestre in modo che permettano di vedere il programma e l'output del programma.

Bene, se tutto funziona abbiamo un bell'elenco di numeri scritti uno sotto l'altro, ma l'allineamento a sinistra non è carino: torniamo al programma e modifichiamolo in modo da riservare 5 spazi per questi numeri:

```
for num in range(101):
    print "%5s" % num
```

Con il solito `<F5>` verifichiamo che il programma faccia quello che vogliamo noi. Aggiungere i quadrati e i cubi non è una novità dato che questo problema è già stato risolto quando abbiamo visto il ciclo `for`. Magari potrà servire qualche prova per aggiustare gli allineamenti:

```
for num in range(101):
    print "%5s" % (num, num*num, num*num*num)
```

quando eseguiamo questo programma otteniamo il seguente deludente risultato:

```
Traceback (most recent call last):
  File "/dati/daniele/06-07/scuola/materiali/informatica/python/primo.py",
  line 8, in -toplevel-
    print "%5s" % (num, num*num, num*num*num)
TypeError: not all arguments converted during string formatting
```

Se vogliamo imparare a programmare dobbiamo abituarci a queste sorprese e imparare a leggere i messaggi di errore. Nel precedente messaggio ci sono diverse informazioni: il nome del file, la riga incriminata e il tipo di errore.

Possiamo tradurre il messaggio così:

```
"Nella linea 8 del file primo.py ci sono dei valori che non sono potuti
entrare  nella stringa formattata."
```

Torniamo a modificare il programma inserendo i segnaposti mancanti:

```
for num in range(101):
    print "%4s  %6s  %8s" % (num, num*num, num*num*num)
```

Questi valori potrebbero andare. Ora dobbiamo inserire altre due colonne e calcolare le radici quadrate e cubiche. Abbiamo già visto che Python senza il supporto di apposite librerie non è in grado di calcolare le radici, quindi dobbiamo modificare il programma in modo che importi la libreria math.

Aggiungiamo prima del ciclo il comando:

```
import math
```

Riassumendo

- Un programma è un documento di testo che contiene le istruzioni che devono essere eseguite.
- Concettualmente dobbiamo distinguere bene la fase di scrittura del programma, dalla fase di esecuzione.
- In pratica conviene scrivere un pezzetto di programma, provarlo, correggerlo, riprovarlo e, quando quel pezzo produce quello che vogliamo noi, aggiungere altre funzionalità.
- Durante la programmazione è inevitabile commettere errori. I messaggi di errore, se letti, sono di grande aiuto.

Prova tu

1. Completa il programma precedente. Come è possibile far calcolare la radice cubica se non c'è una apposita funzione?
2. Aggiungi al programma le istruzioni che stampino anche l'intestazione della tabella.
3. Scrivi il programma "secondo.py" che stampi le lunghezze dei lati dei triangoli rettangoli isosceli con i cateti che variano tra 1 a 10.

1.10 Organizzazione

Come dare un nome a pezzi di programma, definizione e uso di funzioni.

Un programma è una sequenza di istruzioni ma sono passati i tempi in cui un programma era semplicemente una lista numerata di istruzioni. In un buon programma le istruzioni sono raggruppate in blocchi che hanno un senso preciso, una funzione precisa. questi blocchi si

chiamano appunto funzioni (in altri linguaggi si distingue tra funzioni e procedure). Ad ogni funzione possiamo dare un suo nome e ogni volta che definiamo una nuova funzione il linguaggio si amplia di un nuovo comando. Ad es. se definisco una funzione che si chiama "saluta", il linguaggio saprà eseguire il comando "saluta". Una funzione è costituita dalla parola riservata "def" seguita dal nome della funzione, da una coppia di parentesi tonde, dal carattere":" e da un blocco di istruzioni:

```
def <nome>():
    <istruzioni>
```

<nome> è una qualunque parola, ma dobbiamo scegliere una parola che descriva meglio possibile il significato della funzione stessa: il linguaggio di programmazione serve innanzitutto a rendere comprensibile un programma agli umani! <istruzioni> è una qualsiasi sequenza di istruzioni che può essere costituita a da istruzioni già presenti nel linguaggio o definite da noi come nuove funzioni.

Un programma generalmente non è altro che un insieme di funzioni. Per allenarci scriviamo le procedure che calcolino i primi termini di alcune successioni.

1. Iniziamo creando un nuovo programma: da IDLE, Menu: File - New Window.
2. Poi salviamo il file con il nome successioni.py, Menu: File - Save as.
3. Ora scriviamo le prime righe del programma: i commenti che contengono: data, titolo, autori.

Iniziamo con la successione dei primi 10 numeri triangolari. Questa successione si ottiene partendo da 0 e aggiungendo il successivo numero naturale all'ultimo numero triangolare:

```
0+0, 0+1, 1+2, 3+3, 6+4, ...
```

Serve quindi una variabile che contenga il numero triangolare (all'inizio 0) e un ciclo che percorra i primi numeri naturali (ciclo for). All'interno del ciclo aggiungo al numero triangolare attuale l'attuale numero naturale e stampo il nuovo valore:

```
def triangolari10():
    triangolare=0
    for num in range(10):
        triangolare+=num
        print "%5s" % triangolare
```

Ora proviamo a eseguire il programma: <F5>... Non succede niente! In realtà Pyhon ha lavorato, ma il risultato di questo lavoro non ci appare. Python ha imparato cosa deve fare quando noi gli diciamo di eseguire triangolari10. Per provarlo ci spostiamo nell'ambiente IDLE e diamo il comando:

```
>>> triangolari10()
```

Se non ci vengono segnalati errori, vengono stampati i primi 10 numeri triangolari. Ma se non ci andassero bene i primi 10 numeri della successione, ne volessimo 15 o 20 o 100? Possiamo scrivere un'altra procedura, simile a questa che ci chieda quanti numeri vogliamo stampare. L'istruzione che permette di leggere da tastiera una stringa di caratteri è raw_input([<messaggio>]).

Possiamo anche iniziare la funzione con un'istruzione che stampi un'intestazione:

```
def triangolari():
    print "Successione dei numeri triangolari"
    print
    n=raw_input("Quanti termini della successione? ")
    triangolare=0
    for num in range(n):
        triangolare+=num
        print "%5s" % triangolare
```

Solito <F5> e poi nella shell di IDLE. diamo il comando `triangolari()`, rispondiamo alla richiesta con un numero, ... e leggiamo il messaggio di errore:

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in -toplevel-
    triangolari()
  File "/.../python/successioni.py", line 16, in triangolari
    for num in range(n):
TypeError: an integer is required
```

Ma noi avevamo scritto un numero come mai `range` dice che in `n` non c'è un numero? La funzione `raw_input()` restituisce una stringa, dobbiamo convertirla in numero. Quindi ritorniamo nel programma e modifichiamo la terza riga della funzione in modo che esegua la conversione e inserisca in `n`, non il risultato di `raw_input` ma la conversione in intero del risultato di `raw_input`:

```
n <-- int <-- raw_input
```

In Python:

```
n=int(raw_input("Quanti termini della successione? "))
```

Con questa modifica, se non ci sono altri errori, la funzione produce la successione desiderata.

Riassumendo

- In un programma, le istruzione vanno raggruppate in funzioni (o procedure).
- La sintassi per scrivere una funzione è:

```
    def <nome>():
        <istruzioni>
```

- Bisogna distinguere bene la definizione di una funzione dall'esecuzione della funzione stessa. Quando si definisce (`def ...`) la funzione, Python associa un blocco di codice a un nome. Quando si esegue una funzione, si scrive il suo nome seguito da una coppia di parentesi e Python esegue il codice associato a quel nome.
- Il meccanismo delle funzioni permette di creare delle nuove funzionalità del linguaggio permettendo di collegare un blocco di codice a una parola.

Prova tu

1. Scrivi le funzioni che stampano i primi 10 numeri pari e i primi 10 numeri dispari.

2. Scrivi le funzioni, derivate dalle precedenti, ma che chiedono quanti numeri stampare.
3. Scrivi le funzioni che stampano i numeri quadrati e i numeri esagonali.

1.11 Parametri

Come generalizzare funzioni, come passare valori alle funzioni.

Le funzioni definite dal programmatore sono molto utili per dividere il programma in parti sufficientemente piccole. Ma hanno la possibilità di diventare molto più utili se al nome della funzione aggiungiamo dei parametri. I parametri sono delle variabili, dei nomi scritti tra le parentesi che seguono il nome della funzione. Ai parametri vengono associati degli oggetti quando la funzione stessa viene chiamata.

Riprendiamo il programma successioni. Salviamolo cambiandogli il nome (`Menu: File - Save as`), chiamiamolo `successioni02.py`. Aggiustiamo le righe di intestazione del programma (la data e il titolo). Modifichiamo ora la funzione triangolari aggiungendo tra le parentesi il nome di una variabile: "n". Togliamo anche le prime tre righe in modo da avere:

```
def triangolari(n):
    triangolare=0
    for num in range(n):
        triangolare+=num
        print "%5s" % triangolare
```

Il codice ritorna ad essere essenziale e pulito. Proviamolo: `<F5>` e poi nella shell di `IDLE`:

```
>>> triangolari()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in -toplevel-
    triangolari()
TypeError: triangolari() takes exactly 1 argument (0 given)
```

Il risultato non è proprio quello che ci aspettavamo! Il messaggio ci segnala che `triangolari` si aspetta di ricevere un argomento mentre non ne ha ricevuto neanche uno, ciò non gli va bene e Python protesta. In pratica, alla variabile `n` (parametro) che nella definizione della funzione è messa tra parentesi, non è stato associato nessun oggetto (argomento) e quindi triangolari non può andare avanti. Come dare un valore al parametro `n`? Al momento della chiamata della funzione, tra le parentesi possiamo inserire degli argomenti e questi verranno associati ai parametri.

```
>>> triangolari(5)
    0
    1
    3
    6
   10
```

I parametri ci permettono di scrivere procedure più flessibili che utilizzano variabili il cui valore viene definito al momento della chiamata della funzione stessa.

Affrontiamo un altro problema: proviamo a scrivere procedure che calcolino aree e perimetri delle figure piane studiate alle medie. Per iniziare voglio una procedura che, dati base e altezza di un triangolo, ne stampi l'area. Poiché i dati sono 2, avrà bisogno di 2 parametri, e dato che devono contenere la misura della base e la misura dell'altezza conviene chiamarle una "base" e l'altra "altezza" (ma no!). Nota: i nomi delle funzioni e delle variabili devono aver senso per noi, per il computer un nome o l'altro è lo stesso.

Dato che iniziamo un nuovo progetto, creiamo un nuovo file e intestiamolo con i soliti commenti, poi iniziamo a riempirlo con le funzioni necessarie. Ad esempio:

```
def areatriangoloBH(base, altezza):
    print "Area del triangolo"
    print "base = %s" % base
    print "altezza = %s" % altezza
    print "area = %s" % (base*altezza/2.)
```

Alcune osservazioni:

1. Dato che ci sono più modi per calcolare l'area del triangolo, il nome della funzione contiene quelle due strane lettere: `BH` che indicano base e altezza.
2. Nell'ultima linea della funzione bisogna prestare attenzione ad un paio di particolari.
   (a) Pur essendoci un solo elemento da inserire nella stringa, le parentesi sono necessarie pena uno strano errore dovuto a questioni di precedenza degli operatori: "%" viene eseguito prima di "*".
   (b) dopo il 2 è necessario un punto, perché?

Un'altra "famosa" formula per calcolare l'area del triangolo è quella che parte dai 3 lati: la formula di Erone.

Per chi se l'è dimenticata ecco la famosa formula dove `p` è il semi perimetro e `l1`, `l2`, `l3` sono i tre lati:

$$S = \sqrt{p(p - l1)(p - l2)(p - l3)}$$

Riassumendo

- I *parametri* sono delle variabili di proprietà delle funzioni.
- Ogni parametro ha un nome che viene deciso quando si definisce la funzione.
- Il contenuto dei parametri si chiama *argomento* e viene deciso quando viene chiamata la funzione che li contiene.
- I parametri permettono di scrivere funzioni che hanno comportamenti diversi, decisi al momento della chiamata.
- Una funzione può chiamare un'altra funzione, che a sua volta può chiamarne un'altra, e così via.

Prova tu

1. Modifica le funzioni copiate da successioni.py in modo che accettino un parametro e cancella quelle divenute superflue.

2. Scrivi la funzione che calcola l'area del triangolo con la formula di Erone.
3. Prova la precedente funzione finché non dà un errore, quali valori possono dare errore, perché?
4. Scrivi le funzioni che calcolano i perimetri dei poligoni che conosci.
5. Scrivi le funzioni che calcolano le aree dei quadrilateri notevoli.

1.12 Nuove operazioni

Come scrivere funzioni con un risultato.

Abbiamo scritto delle funzioni che calcolano e stampano il risultato di certe formule, ma i linguaggi di programmazione permettono anche di scrivere dei blocchi di codice che si comportano come delle operazioni cioè funzioni che danno un risultato. I linguaggi di programmazione permettono perciò di creare delle nuove operazioni. L'istruzione che permette di creare nuove operazioni è l'istruzione `return`. Questa istruzione ha due effetti: termina una funzione e restituisce un risultato a chi l'ha chiamata.

Vediamo un esempio, possiamo scrivere l'operazione che restituisce la media di due numeri. Iniziamo un nuovo programma, `menu -File-New window`, solite righe di commento, poi scriviamo la funzione:

```
def media(n1, n2):
    return (n1+n2)/2.
```

Solito <F5> e poi nella shell di IDLE:

```
>>> print media(12, 18)
15.0
```

Il fatto di usare delle funzioni che restituiscono un valore è importante in informatica, permette di separare la parte di calcolo da quella di `input` e `output` dei dati.

Riprendiamo il calcolo dell'area di un triangolo dati i tre lati, possiamo scrivere una funzione che calcoli la formula di Erone:

```
def erone(l1, l2, l3):
    p=(l1+l2+l3)/2.
    return (p*(p-l1)*(p-l2)*(p-l3))**0.5
```

Osservazioni:

1. Nella formula `p=(l1+l2+l3)/2.` le parentesi e il punto dopo il 2 sono necessari, perché?
2. Questa volta non ho utilizzato la libreria `math` per calcolare la radice quadrata, ma ho usato un trucco matematico: la radice quadrata di un numero è uguale a una potenza che ha per esponente il valore ½ o 0.5 che è equivalente.
3. Nei fogli di calcolo e in altri linguaggi il simbolo della potenza è "^", in Python è una coppia di asterischi "**".

Ora la funzione areatriangoloABC diventa:

```
def areatriangoloABC(l1, l2, l3):
    """Calcolo dell'area con la formula di Erone."""
    # Immissione dati
    print "Area del triangolo"
    print "lato1 = %s" % l1
    print "lato2 = %s" % l2
    print "lato3 = %s" % l3
    # Calcolo
    area=erone(l1, l2, l3)
    # Visualizzazione risultati
    print "area = %s" % area
```

In questo modo abbiamo realizzato una separazione tra la funzione che realizza la comunicazione con l'utente e la funzione che realizza il calcolo. Questa separazione è molto importante nei programmi "seri". Le funzioni permettono un tipo di programmazione molto sintetica e pulita. Ci sono linguaggi di programmazione che usano pesantemente questa tecnica, sono detti linguaggi funzionali. Python permette anche una programmazione di tipo funzionale.

Supponiamo di dover calcolare la media dell'area di due triangoli. La traduzione in Python diventa quasi letterale:

```
def mediatri(a1, a2, a3, b1, b2, b3):
  return media(erone(a1, a2, a3), erone(b1, b2, b3))
```

e nella shell di IDLE:

```
>>> print mediatri(3, 4, 5, 13, 14, 15)
45.0
```

Come funziona? `mediatri` riceve 6 numeri, i lati dei 2 triangoli, chiama la funzione `media` e le passa due valori i risultati di `erone` con i primi 3 lati e `erone` con gli altri 3. Le funzioni `erone` calcolano le due aree, la funzione `media` ne calcola la media e la funzione `mediatri` restituisce il risultato.

Python permette anche di costruire funzioni che danno più risultati. Questo può risultare utile in matematica, ad esempio il punto medio `M` di un segmento di estremi `A=(xa, ya)` e `B=(xb, e yb)` ha per coordinate `xm=(xa+xb)/2.` e `ym=(ya+yb)/2.`. In Python si può scrivere la funzione:

```
def puntomedio(xa, ya, xb, yb):
  return (xa+xb)/2., (ya+yb)/2.
```

e nella shell di IDLE:

```
>>> print puntomedio(2, 5, -6, 1)
(-2.0, 3.0)
```

Riassumendo

- È possibile scrivere funzioni che restituiscono un risultato.
- Una funzione termina quando non ci sono più istruzioni da eseguire o quando viene eseguita l'istruzione `return`.

- Il comando che termina l'esecuzione di una funzione fornendo il risultato è `return <espressione[,..]>`.
- Una funzione può restituire più di un risultato, in questo caso i risultati vanno elencati dopo l'istruzione `return`, separati da virgole.
- Python permette una programmazione funzionale.

Prova tu

1. Riscrivi il calcolo dell'area dei poligoni usando delle funzioni.
2. Scrivi la funzione che calcola la media tra 3 numeri.
3. Riscrivi la funzione `puntomedio()` in modo che usi la funzione `media`.
4. Scrivi la funzione che restituisce coefficiente angolare e termine noto della retta passante per due punti.
5. Scrivi la funzione che restituisce coefficiente angolare e termine noto dell'asse di un segmento.
6. Scrivi la funzione che effettua la traslazione di un punto nel piano cartesiano.
7. Scrivi le funzioni che effettuano il ribaltamento di un punto rispetto agli assi e all'origine.

1.13 Operare scelte

Come eseguire diverse porzioni di codice a seconda del risultato di un'espressione, come usare l'istruzione di scelta o di selezione.

I programmi e le funzioni scritti finora eseguono tutte le istruzioni che vengono scritte, ma i linguaggi di programmazione danno anche la possibilità di eseguire un gruppo di istruzioni o un altro in base a determinate condizioni. Permettono cioè di descrivere algoritmi che svolgono il seguente compito:

```
se è vera una certa condizione
    esegui qualcosa
altrimenti
    esegui qualcos'altro
```

Il principale comando che permette di operare delle scelte è `if` e la sua sintassi è una delle seguenti.

```
if <condizione>:
    <istruzioni>
```

oppure:

```
if <condizione>:
    <istruzioni 1>
else:
    <istruzioni 2>
```

Nel calcolo dell'area del triangolo dati i lati, la formula di Erone funziona solo se il triangolo può essere costruito. In ogni triangolo un lato non può essere maggiore della somma degli altri due né minore della loro differenza. Queste due affermazioni vengono dette disuguaglianze triangolari. Se i lati non rispettano le disuguaglianze triangolari allora la funzione termina con un errore infatti, in questo caso, l'espressione all'interno della radice assume valore negativo.

È possibile modificare la funzione in modo che restituisca un valore particolare nel caso i tre numeri non possano essere le misure dei lati di un triangolo. Possiamo quindi modificare la funzione `erone` in modo che controlli che ogni lato sia minore della somma degli altri due. Calcoliamo quindi le somme delle coppie di lati e confrontiamole con il terzo lato; se tutti questi confronti danno esito positivo calcoliamo l'area, altrimenti restituiamo il valore None. La funzione `erone()` può essere scritta così:

```
def erone(lato1, lato2, lato3):
    """Restituisce l'area di un triangolo dati i tre lati
      utilizzando la Formula di Erone."""
    if (lato3<=lato1+lato2 and
        lato2<=lato1+lato3 and
        lato1<=lato2+lato3):
        p=(lato1+lato2+lato3)/2.
        return (p*(p-lato1)*(p-lato2)*(p-lato3))**0.5
    else:
        return None
```

Ora scriviamo una funzione che legga dall'utente la lunghezza dei tre lati del triangolo e ne stampi l'area:

```
def areatriangolo():
    """Calcolo dell'area con la formula di Erone."""
    # Inserimento dati
    print "Area del triangolo"
    lato1=float(raw_input("primo lato = "))
    lato2=float(raw_input("secondo lato = "))
    lato3=float(raw_input("terzo lato = "))
    # Calcolo
    area=erone(lato1, lato2, lato3)
    # Visualizzazione risultati
    if area!=None:
        print "area = %s" % area
    else:
        print "Nessun triangolo può avere questi tre lati!"
```

Proviamo la funzione nella `shell` con diversi valori degli argomenti.

Affrontiamo un altro problema. Proviamo a scrivere una funzione che riceva 2 numeri e dia come risultato il più piccolo. Con un linguaggio quasi naturale: i due numeri sono n1 e n2:

```
se n1 è minore di n2
    allora il risultato è n1
    altrimenti il risultato è n2
```

La traduzione in Python è abbastanza semplice e viene lasciata al lettore.

Riassumendo

- È possibile, in base a una condizione eseguire delle istruzioni o delle altre.
- Questa struttura di controllo si chiama "selezione".
- La sintassi dell'istruzione `if` è:

```
if <condizione1>:
    <istruzioni 1>
[elif <condizione2>:
    <istruzioni 2>
[...]]
[else:
    <istruzioni 3>]
```

- Le parentesi quadre indicano porzioni facoltative di codice.
- Le clausole `elif` e `else` possono non esserci.

Prova tu

1. Scrivi la funzione massimo che dà come risultato il massimo tra 2 numeri.
2. Scrivi la funzione che, usando la funzione massimo, legge 2 numeri e stampa il maggiore dei due.
3. Scrivi le funzioni min3 e max3 che restituiscono rispettivamente il minimo e il massimo tra 3 numeri.
4. Scrivi la funzione ordina2 che dati due numeri li restituisce disposti in ordine, prima il più piccolo e poi il più grande.
5. Scrivi la funzione ordina3, analoga alla precedente, ma che lavora su 3 numeri.
6. Riscrivi la funzione che calcola il coefficiente angolare di una retta, dati 2 punti, considerando anche il caso in cui i due punti abbiano la stessa ascissa.
7. Scrivi una funzione che "forza" un valore all'interno di un massimo e un minimo: taglia(50, 68, 100)->68; taglia(50, 27, 100)->50

1.14 Sequenze e cicli

Come trattare sequenze di dati.

Nei capitoli precedenti abbiamo visto alcuni tipi di dati, `int`, `float`, `str` e un modo per ripetere le istruzioni usando l'istruzione `for`. In questo capitolo approfondiamo la conoscenza delle stringhe e dell'istruzione `for`, vedendo come sono collegati tra di loro. Come già visto una stringa è una sequenza di caratteri e in ogni sequenza, si può:

- estrarne un elemento,
- estrarne uno alla volta tutti gli elementi,
- estrarne una porzione (una fetta),

- ottenere il numero di elementi che la compongono.

Costruiamo una stringa e stampiamo la sua lunghezza, cioè il numero di elementi che la compongono:

```
>>> s="Pippo Pluto Paperino"
>>> print(s)
Pippo Pluto Paperino
>>> print(len(s))
20
```

Se vogliamo stampare un suo carattere possiamo applicare alla sequenza un indice:

```
>>> print(s[1])
i
```

Possiamo anche dire a Python di stamparne più di uno separati da uno spazio, magari iniziando a contare dall'ultimo elemento:

```
>>> print s[-1], s[-4], s[-10]
o r o
```

Ed ora vediamo come il ciclo `for` è collegato con le sequenze. L'uso già visto nel capitolo sulla ripetizione di istruzioni è solo una delle possibilità dell'istruzione `for`. In generale questa istruzione permette di scorrere gli elementi di una sequenza. Come esempio, stampiamo una sotto l'altra le lettere di una parola:

```
>>> for carattere in "Casa": print carattere

C
a
s
a
```

Notare l'uso della virgola nell'istruzione. In un nuovo file (`stringhe.py`) scriviamo la procedura per stampare una parola triangolare: l'intera parola, la parola senza la prima lettera, poi senza la seconda, e così via:

```
def triangolo(s):
    for indice in range(len(s)):
        print s[indice:]
```

e, nella shell di `IDLE`, la proviamo:

```
>>> triangolo("scala")
scala
cala
ala
la
a
```

È possibile controllare se un carattere appartiene ad una stringa con l'operatore `in`:

```
>>> 'u' in 'trallallà'
False
>>> 'a' in 'trallallà'
True
```

Se volessimo contare le vocali presenti in una stringa o più in generale quante sono le lettere che appartengono ad un determinato insieme possiamo creare una funzione che riceva due argomenti: la stringa in cui contare i caratteri e una stringa che contiene i caratteri da contare:

```
def contacaratteri(stringa, caratteri):
    """Restituisce il numero degli elementi di stringa presenti in
       caratteri."""
    contatore = 0
    for c in stringa:
        if c in caratteri:
            contatore += 1
    return contatore

>>> print(contacaratteri('ambarabà cicì cocò', 'aeiou'))
5
```

Riassumendo

- Le stringhe sono degli oggetti - sequenza, vuol dire che sono viste da Python come una sequenza ordinata di elementi. Nel caso delle stringhe, ovviamente, gli elementi sono caratteri.
- Il primo elemento di una sequenza ha sempre indice 0, l'ultimo ha indice -1.
- È possibile estrarre un elemento di una sequenza con la sintassi: `<sequenza>[<indice>]`.
- L'indice può essere un numero positivo (il conteggio parte dall'inizio) o negativo (il conteggio parte dalla fine). Se l'indice eccede i limiti della sequenza viene sollevato un errore.
- L'istruzione `for` permette di scorrere gli elementi di una sequenza, la sua sintassi è:

```
    for <variabile> in <sequenza>
        <istruzioni>
```

Prova tu

1. Scrivi la procedura verticale(s) che stampa i caratteri di una stringa uno sotto l'altro.
2. Scrivi la procedura triangoloinverso(s) che stampa l'ultimo carattere di una stringa, poi gli ultimi due e così via fino a stampare tutta la stringa.
3. Scrivi la funzione che, data una stringa, stampi le sue lettere saltando gli spazi.
4. Scrivi la funzione che, data una stringa, stampi le sue lettere saltando le vocali.

1.15 Tuple

Come trattare sequenze di oggetti vari.

Separare i dati dalle espressioni o dagli algoritmi che li usano è uno dei metodi fondamentali dell'informatica. Questo consente di accedere agli stessi dati in momenti diversi e per scopi (operazioni o programmi) diversi. Oppure consente di di utilizzare gli stessi programmi su dati diversi. Nei linguaggi di programmazione, di solito, un nome che fa riferimento ad un valore viene detto "variabile", perché il nome, in momenti diversi, può fare riferimento a valori diversi. Le variabili in Python sono dunque degli identificatori collegati con oggetti. Alcuni tipi di oggetti li abbiamo già incontrati: numeri interi (`int`), numeri in virgola mobile (`float`), stringhe (`str`). In questo e nei prossimi capitoli parleremo di: tuple (`tuple`), liste (`list`) e dizionari (`dict`), ... in realtà ogni cosa in Python è un oggetto.

Iniziamo dalle `tuple`.

Abbiamo visto che una `stringa` è una sequenza di caratteri, ma se abbiamo bisogno di una sequenza di oggetti diversi dai caratteri, dobbiamo utilizzare altri strumenti. Supponiamo di avere una centralina atmosferica che deve trasmettere i seguenti dati:

- la temperatura,
- l'umidità,
- la pressione atmosferica,
- la velocità e la direzione del vento,
- lo stato del cielo.

Questi dati potrebbero essere inseriti in una `tupla`:

```
>>> meteo = (23.5, 76, 1042, (2.5, "NE"), "cloud")
```

Alcune osservazioni:

1. Una `tupla` è una sequenza che può contenere oggetti diversi, anche altre `tuple`.
2. Si possono estrarre gli elementi di una sequenza usando gli indici con la sintassi: `<sequenza>[<indice>]`
3. È Importante ricordare che tutte le sequenze, in Python partono dall'elemento 0, cioè il primo elemento di una sequenza ha sempre indice 0, il secondo ha indice 1 e così via.
4. Possiamo anche estrarre gli elementi riferendoci alla fine della sequenza: l'ultimo elemento ha indice -1, il penultimo -2 e così via.

Il primo elemento della `tupla` precedente è un numero in virgola mobile, poi ci sono due numeri interi, il quarto elemento è una `tupla` formata da un numero e una stringa e il quinto è una stringa.

Posso estrarre dall'oggetto `meteo` i dati relativi al vento con l'istruzione:

```
>>> print(meteo[3])
(2.5, 'NE')
```

Per stampare temperatura, umidità e pressione:

```
>>> print(meteo[:3])
(23.5, 76, 1042)
```

Se volessi solo la direzione del vento basta che prenda il secondo elemento del quarto elemento di `meteo` (ricordiamoci che l'indicizzazione parte sempre da 0):

```
>>> print(meteo[3][1])
NE
```

Vediamo comunque un altro uso delle `tuple`. Supponiamo di avere inserito in una `tupla` il nome l'età e l'indirizzo di una persona:

```
>>> omo = ("Mario Rossi", 45, "via Verdi, 4")
```

Possiamo stampare questi dati per ottenere una specie di etichetta:

```
    >>> print("Sig.", omo[0], "\nabitante in", omo[2], "\ndi anni", omo[1])
    Sig. Mario Rossi
    abitante in via Verdi, 4
    di anni 45
```

Lo stesso effetto lo possiamo ottenere usando la formattazione di stringhe:

```
>>> print("Sig. {0}\nabitante in {1}\ndi anni{2}".format(omo))
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print("Sig. {0}\nabitante in {1}\ndi anni{2}".format(omo))
IndexError: tuple index out of range
```

Accidenti, così non funziona infatti `format` vuole avere separatamente gli oggetti che deve inserire nella stringa, per disfare una tupla nelle sue varie componenti basta far precedere un asterisco al nome collegato alla `tupla`:

```
>>> print("Sig. {0}\nabitante in {1}\ndi anni{2}".format(*omo))
Sig. Mario Rossi
abitante in 45
di annivia Verdi, 4
```

C'è ancora qualcosa che non va, come modificare l'ordine delle variabili?

Riassumendo

- Le liste e le tuple sono contenitori di oggetti. L'indicizzazione e l'affettamento, che abbiamo visto per le stringhe, funzionano allo stesso modo per tutte le sequenze.
- Le liste sono sequenze modificabili, le tuple no.
- Il metodo append permette di aggiungere elementi alla fine delle liste. La sintassi è:

```
    <lista>.append(<oggetto>).
```

- In Python esiste anche una sintassi particolare per costruire liste.

Prova tu

1. Scrivi altre funzioni statistiche per liste di numeri: massimo, scarto quadratico medio, ...
2. Integra queste nuove funzioni nella procedura statistica.
3. Scrivi una funzione che stampi gli elementi di una lista uno sotto l'altro.
4. Scrivi una funzione che, data una lista di numeri e un valore, stampi solo i numeri maggiori di quel valore.
5. Scrivi una funzione che, data una lista di numeri e un valore, restituisca una lista formata solo dagli elementi della lista maggiori di quel valore.

1.16 Liste

Come trattare sequenze modificabili di oggetti vari.

Supponiamo che un sistema di sorveglianza, attraverso di sensori tenga sotto controllo 6 porte. Vogliamo memorizzare in 6 variabili lo stato delle porte. Ma questa soluzione si dimostra poco flessibile. Possiamo anche raggruppare in un'unica variabile lo stato di tutte le porte. Potremmo usare una `tupla` come visto nel capitolo precedente, ma vogliamo anche che quando il programma si accorge che una porta è stata aperta o chiusa, modifichi il suo stato nella variabile. Le `tuple` sono delle sequenze immutabili quindi non sono adatte come soluzione di questo problema. Le `tuple`, una volta create, non possono essere modificate. In questo caso dobbiamo usare un oggetto `lista`. Se inizialmente tutte le porte sono chiuse avremo:

```
>>> porte = ['c', 'c', 'c', 'c', 'c', 'c']
>>> print(porte)
['c', 'c', 'c', 'c', 'c', 'c']
```

Possiamo cambiare lo stato di una porta, ad esempio la prima:

```
>>> porte[0] = 'a'
>>> print(porte)
['a', 'c', 'c', 'c', 'c', 'c']
```

Il comando precedente può essere interpretato come: "la porta di posizione 0" è stata aperta. In modo analogo possiamo aprire un'altra porta o chiuderne una:

```
>>> porte[3] = 'a'
>>> print(porte)
['a', 'c', 'c', 'a', 'c', 'c']
>>> porte[0] = 'c'
>>> print(porte)
['c', 'c', 'c', 'a', 'c', 'c']
```

SON QUI SON QUI SON QUI

Separare i dati dalle espressioni o dagli algoritmi che li usano è uno dei metodi fondamentali dell'informatica. Questo consente di accedere agli stessi dati in momenti diversi e per scopi (operazioni o programmi) diversi. Oppure consente di di utilizzare gli stessi programmi su dati diversi. Nei linguaggi di programmazione, di solito, un nome che fa riferimento ad un valore

viene detto "variabile", perché il nome, in momenti diversi, può fare riferimento a valori diversi. Le variabili in Python sono dunque degli identificatori collegati con oggetti. Alcuni tipi di oggetti li abbiamo già incontrati: numeri interi (`int`), numeri in virgola mobile (`float`), stringhe (`str`). Ce ne sono altri che impareremo a usare più avanti: liste (`list`), tuple (`tuple`), dizionari (`dict`), ... in realtà ogni cosa in Python è un oggetto.

Vediamo comunque subito un primo banale uso delle liste:

```
>>> omo=["Mario Rossi", 45, "via Verdi, 4"]
>>> print("Sig.", omo[0], "\nabitante in", omo[2], "\ndi anni", omo[1])
Sig. Mario Rossi
abitante in via Verdi, 4
di anni 45
```

Alcune osservazioni:

1. Una lista può contenere oggetti diversi, anche altre liste. Nel caso precedente, il primo e l'ultimo elemento della lista sono stringhe il secondo è un numero intero.
2. Si possono estrarre gli elementi di una lista usando gli indici con la sintassi: `<lista>[<indice>]`
3. È Importante ricordare che le liste partono dall'elemento 0, cioè il primo elemento di una lista ha sempre indice 0, il secondo ha indice 1 e così via.
4. Possiamo anche estrarre gli elementi riferendoci alla fine della lista: l'ultimo elemento ha indice -1, il penultimo -2 e così via.

Python mette a disposizione strumenti per trattare sequenze di oggetti qualsiasi in modo analogo a come tratta le stringhe che sono sequenze di caratteri. Gli oggetti liste e le tuple sono sequenze ordinate di altri oggetti che possono essere numeri, stringhe o anche liste o tuple. Una lista può essere costruita ponendo gli oggetti, separati da virgole, tra una coppia di parentesi quadre. Una tupla è costruita ponendo gli oggetti, sempre separati da virgole, tra una coppia di parentesi tonde. Esempi:

```
>>> lista=[3, "pippo", ["sotto", "lista"], 3.14]
>>> tupla=(["multipli", "di", 7], 0, 7, 14, "ecc")
>>> print len(lista)
4
>>> print len(tupla)
5
```

Alla parola lista è associata una lista di 4 elementi: un intero, una stringa, una lista e un numero in virgola mobile. Alla parola tupla è associata una tupla di 5 elementi: una lista, 3 interi e una stringa.

Tutto quello che abbiamo detto sulle stringhe come sequenze, i metodi per estrarre elementi o fette, vale anche per le liste e per le tuple. Ad es. se voglio estrarre l'ultimo elemento da `lista` e i tre interi da `tupla` posso scrivere:

```
>>> print lista[-1]
3.14
>>> print tupla[1:4]
(0, 7, 14)
```

Bisogna ricordare che in tutte le sequenze l'indice parte da `0`. Liste e tuple sono molto simili tra di loro, la differenza è piuttosto tecnica: le liste sono oggetti modificabili mentre le tuple, come le stringhe, una volta create, non possono essere modificate. Le operazioni che modificano l'oggetto possono essere compiute sulle liste ma non su stringhe e tuple. Nelle liste è possibile eliminare un elemento o inserirlo in un certo punto:

```
>>> del lista[1]
>>> print lista
[3, ['sotto', 'lista'], 3.1400000000000001]
>>> lista.insert(2, "Mario")
>>> print lista
[3, ['sotto', 'lista'], 'Mario', 3.1400000000000001]
```

Ma l'operazione che useremo più spesso sarà quella di costruire una lista. Si può partire da una lista vuota e aggiungere elementi alla fine. Ad esempio se voglio una lista di numeri quadrati incomincio assegnando alla parola `quadrati` una lista vuota e poi, in un ciclo, con il metodo `append` aggiungo alla lista quanti quadrati voglio:

```
>>> quadrati=[]
>>> for num in range(10): quadrati.append(num*num)

>>> print quadrati
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

C'è anche un altro modo per ottenere lo stesso risultato usando la sintassi della costruzione di liste. Ad esempio se voglio una lista con i primi 10 numeri cubi posso usare un meccanismo analogo al precedente, ma con una scrittura più sintetica:

```
>>> cubi=[num*num*num for num in range(10)]
>>> print cubi
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Scriviamo un programma che stampi alcuni dati statistici sui numeri contenuti in una lista. La somma degli elementi di una lista di numeri si può ottenere inizializzando a `0` una variabile e poi aggiungendo, a questa, ogni elemento della lista:

```
def somma(lista):
    """Somma gli elementi di una lista di numeri."""
    risultato=0
    for n in lista: risultato+=n
    return risultato
```

Realizzata la funzione somma, trovare la media è banale:

```
def media(lista):
    return somma(lista)/float(len(lista))
```

Si possono realizzare diverse altre funzioni statistiche e una procedura che riceva come parametro una lista e stampi i risultati delle funzioni statistiche che abbiamo creato:

```
def statistica(lista):
    print "La lista è: %s" % lista
```

```
    print "Media dei valori: %s" % media(lista)
    print "Valore minimo %s" % minimo(lista)
```

<F5> e poi nella shell di IDLE la proviamo:

```
>>> statistica([23, 65, 73, 13, 33, 95, 47, 83, 72])
La lista è: [23, 65, 73, 13, 33, 95, 47, 83, 72]
Media dei valori: 56.0
Valore minimo 13
```

Riassumendo

- Le liste e le tuple sono contenitori di oggetti. L'indicizzazione e l'affettamento, che abbiamo visto per le stringhe, funzionano allo stesso modo per tutte le sequenze.
- Le liste sono sequenze modificabili, le tuple no.
- Il metodo append permette di aggiungere elementi alla fine delle liste. La sintassi è:

```
    <lista>.append(<oggetto>).
```

- In Python esiste anche una sintassi particolare per costruire liste.

Prova tu

1. Scrivi altre funzioni statistiche per liste di numeri: massimo, scarto quadratico medio, ...
2. Integra queste nuove funzioni nella procedura statistica.
3. Scrivi una funzione che stampi gli elementi di una lista uno sotto l'altro.
4. Scrivi una funzione che, data una lista di numeri e un valore, stampi solo i numeri maggiori di quel valore.
5. Scrivi una funzione che, data una lista di numeri e un valore, restituisca una lista formata solo dagli elementi della lista maggiori di quel valore.

1.17 Il ciclo while

Come ripetere un ciclo un numero di volte non predeterminato.

Abbiamo visto come ripetere blocchi di istruzioni, ma con l'istruzione `for` il numero di ripetizioni deve essere noto prima che inizi il ciclo. Non sempre ciò è possibile. Supponiamo di voler leggere dei numeri dalla tastiera e continuare a farlo fin quando l'utente non immette un numero convenuto. Il programma non può sapere, all'inizio del ciclo, se l'utente immetterà 3, 5 o 5000 numeri. I linguaggi di programmazione mettono a disposizione un costrutto che permette di risolvere questo problema: il ciclo `while`. In Pyhton la sintassi è:

```
while <condizione>:
    <istruzioni>
```

Scriviamo la procedura che legge numeri scritti dall'utente, ne calcola la somma, e si ferma quando viene immesso il numero 0. La procedura deve:

- predisporre la variabile che conterrà la somma inizializzandola a 0.
- eseguire un ciclo in cui legge un numero e lo aggiunge alla somma fin quando il numero è diverso da 0.

Una prima procedura potrebbe essere:

```
def sommanumeri():
    somma = 0
    while numero <> 0:
        numero = int(raw_input("scrivi un numero: "))
        somma += numero
    print "la somma dei numeri immessi è: %s" % somma
```

Alcune osservazioni:

1. `somma = 0` è messo prima del ciclo altrimenti verrebbe continuamente azzerato e non memorizzerebbe la somma dei numeri.
2. L'istruzione `print` è fuori dal ciclo `while` infatti non è indentata come le altre istruzioni del blocco di `while`, quindi verrà eseguita quando il ciclo termina.
3. Prima di mettere in numero quello che ho letto dalla tastiera, devo convertirlo in intero.

Quando proviamo la procedura otteniamo un errore:

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in -toplevel-
    sommanumeri()
  File "/sorgenti/while.py", line 9, in sommanumeri
    while numero<>0:
UnboundLocalError: local variable 'numero' referenced before assignment
```

Questo messaggio ci dice che nella linea 9 del file `while.py`, all'interno della funzione `sommanumeri()` e precisamente nell'istruzione `while numero <> 0:`, Pyhton dovrebbe trovare qualcosa dentro la variabile numero, variabile che non è ancora stata definita, quindi non sa come andare avanti. Mettiamo in `numero` un valore a fantasia, basta che sia diverso da `0` altrimenti il ciclo non viene eseguito neppure una volta. Poi proviamo la procedura:

```
def sommanumeri():
    somma = 0
    numero = 2652
    while numero <> 0:
        numero = int(raw_input("scrivi un numero: "))
        somma += numero
    print "la somma dei numeri immessi è: %s" % somma

>>> sommanumeri()
scrivi un numero: 4
scrivi un numero: 5
scrivi un numero: 6
scrivi un numero: 0
la somma dei numeri immessi è: 15
```

Funziona, ma non è scritta bene:

1. `numero` deve essere inizializzato con un numero che poi non viene usato.
2. il numero immesso convenzionalmente per terminare il ciclo viene usato. Fin che è 0 e finché eseguiamo addizioni, non dà fastidio, ma se il numero convenzionale forre 999 o se stessimo eseguendo moltiplicazioni?

Riporto, di seguito due possibili soluzioni:

```
def sommanumeri2():
    somma = 0
    numero = int(raw_input("scrivi il primo numero: "))
    while numero <> 0:
        somma += numero
        numero = int(raw_input("scrivi un altro numero: "))
    print "la somma dei numeri immessi è: %s" % somma

def sommanumeri3():
    somma = 0
    while True:
        numero = int(raw_input("scrivi un numero: "))
        if numero == 0: break
        somma += numero
    print "la somma dei numeri immessi è: %s" % somma
```

In quest'ultimo caso la condizione che comanda la ripetizione del ciclo è sempre vera quindi il ciclo non termina naturalmente, ma continua all'infinito. All'interno del ciclo c'è però un'istruzione di selezione che, in certe condizioni, segue l'istruzione `break`. L'istruzione `break` fa terminare immediatamente il ciclo in cui si trova.

Riassumendo

- Il ciclo `while` permette di ripetere un blocco di istruzioni fin quando è vera una certa condizione. La sintassi è:

```
    while <condizione>:
        <istruzioni>
```

- Il ciclo `while` si usa quando prima di avviare il ciclo non è possibile conoscere per quante volte dovrà essere ripetuto il blocco di istruzioni.
- In certi casi è comodo realizzare un ciclo infinito e farlo terminare quando si determinano determinate condizioni con l'istruzione `break`.

Prova tu

1. Scrivi la procedura che stampa il prodotto di numeri immessi dall'utente.
2. Scrivi le funzioni che restituiscono la somma e il prodotto di numeri immessi dall'utente.
3. Scrivi la procedura che legge numeri finquando sono maggiori di 0 e poi stampa la media, il massimo e il minimo dei numeri immessi.

1.18 Funzioni ricorsive

Funzioni che definiscono sé stesse.

Abbiamo visto che una volta scritta una funzione, questa può essere chiamata da altre funzioni. Il linguaggio è costruito in modo tale che una funzione può chiamare anche sé stessa. Funzioni che che chiamano sé stesse si dicono ricorsive. Ad esempio la scalinata di Giacobbe può essere definita come uno scalino seguito da una scalinata di giacobbe. In Python:

```
def scalinata_di_Giacobbe():
    print "Scalino"
    scalinata_di_Giacobbe()
```

A prima vista si potrebbe pensare che questa procedura stampi una volta "Scalino" o al massimo 2 volte, invece continua a stampare "Scalino" finché c'è memoria disponibile o viene premuto il tasto <Ctrl-c>.

Come potremmo costruire una scala più umana? una scala di un certo numero di gradini? Prima di scrivere la procedura, scriviamo la definizione:

```
           | se n=0 la scala è già finita
scala(n) = |
           | se n>0 è uno scalino seguito da una scala(n-1)
```

Utilizzando il comando `if` si può controllare la fine del lavoro della procedura. Scriviamo la procedura in un file:

```
def scala(n):
    if n == 0: return
    print "scalino"
    scala(n-1)
```

e proviamo il suo funzionamento: <F5>, poi nella shell di IDLE:

```
>>> scala(3)
scalino
scalino
scalino
```

Anche la procedura che stampa una parola triangolare può essere realizzata usando la ricorsione:

```
def triangolo(s):
    if s == "": return
    print s
    triangolo(s[1:])
```

Diciamo che una procedura ricorsiva è terminale se non vengono eseguite altre istruzioni dopo la chiamata ricorsiva. Le procedure viste sopra sono terminali.

Oltre alle procedure ricorsive, è anche possibile costruire funzioni ricorsive. Molte definizioni aritmetiche sono ricorsive e possono essere tradotte facilmente in funzioni ricorsive. Ad esempio una *potenza* che ha per esponente un numero naturale può essere definita in questo modo:

un numero elevato all'esponente `enne` è uguale a `1` se `enne` è uguale a `0` altrimenti è uguale al numero per la potenza che ha esponente `enne - 1`. Scritto in altro modo:

```
          | se esponente=0 allora la potenza è 1
baseesp = |
          | se esponente>0 allora la potenza è base*baseesp-1
```

La definizione potrà sembrare piuttosto strana, ma tradotta in linguaggio di programmazione funziona:

```
def potenza(base, esponente):
    if esponente == 0:
        return 1
    else:
        return base * potenza(base, esponente - 1)
```

Altro esempio: il *fattoriale* del numero `enne` è `1` se il `enne` è `1`, altrimenti è `enne` per il `fattoriale` di `enne - 1`:

```
             | se n=0 allora il risultato è 1
fatt. di n = |
             | se n>0 allora il risultato è n*fatt. di n-1
```

Un'altra definizione ricorsiva interessante è il metodo di Euclide per determinare il massimo comune divisore tra due numeri: se i due numeri sono uguali il massimo comune divisore è uno dei due, altrimenti è il massimo comune divisore tra il più piccolo e la differenza tra i due numeri:

```
               | se n1=n2 il risultato è n1
macd(n1, n2) = | se n1>n2 il risultato è macd(n1-n2, n2)
               | se n2>n1 il risultato è macd(n2-n1, n1)
```

Data la definizione ricorsiva di una funzione, la sua traduzione in linguaggio Python risulta molto semplice.

Riassumendo

- Le procedure ricorsive con la condizione di terminazione permettono di ripetere dei blocchi di istruzioni.
- Una procedura ricorsiva si dice terminale se non vengono eseguite altre istruzioni dopo la chiamata ricorsiva.
- Molte definizioni matematiche possono essere espresse in forma ricorsiva e sono facilmente traducibili in Python.
- Le definizioni ricorsive sono interessanti perché risolvono un problema utilizzando i termini del problema stesso.

Prova tu

1. Cosa succede se nella procedura triangolo scambi tra di loro l'istruzione `print s` e la chiamata ricorsiva? Prima cerca di immaginarlo, poi controlla scrivendo quest'altra procedura e provandola.

2. Scrivi le funzioni che producono il fattoriale, e il massimo comun divisore.
3. Scrivi una procedura ricorsiva che stampi uno sotto l'altro gli elementi di una lista.
4. Scrivi una funzione ricorsiva che calcoli la somma dei numeri contenuti in una lista.

1.19 File di testo

Come leggere e scrivere su file di testo.

Spesso i programmi devono elaborare una notevole mole di dati. Il luogo migliore dove memorizzare dati è nel disco sotto forma di file, il programma deve dunque essere in grado di leggere e scrivere informazioni su file. In un file i dati possono essere memorizzati in diversi modi, noi vedremo i file di testo. Un file di testo è memorizzato sul disco con un suo nome e contiene stringhe separate le une dalle altre da simboli di fine linea.

Proviamo a realizzare il gioco della maestra. Il programma fa delle domande, legge le risposte e controlla se sono giuste o meno. Le domande e le risposte le memorizziamo in un file di testo:

```
Io scrivo il nome di uno stato, tu devi scrivere il nome della capitale.
Italia        Roma
Francia       Parigi
Gran Bretagna Londra
...
```

Salviamo questo file con il nome "capitali.dat".

Alcune osservazioni:

1. Ho deciso che la consegna viene scritta in un'unica riga, la prima.
2. Nelle righe successive ci sono le domande e le risposte separate da un carattere che non deve essere presente né nelle domande né nelle risposte, io ho scelto il carattere di tabulazione.
3. Il semplice programma che scriveremo non prevede righe vuote o righe di commento.
4. Il file deve essere un file di testo semplice, non deve contenere informazioni relative al formato, ma soltanto caratteri `ascii`.

Ora scriviamo il programma che legge il file:

```
def gioco01(nomefile):
    file_dati = file(nomefile+".dat")
    dati = file_dati.readlines()
    print dati
```

Ora abbiamo una lista di stringhe, stampiamo la prima e avviamo un ciclo sulle altre. In questo ciclo dobbiamo spezzare la stringa in due dove c'è il tabulatore. La prima parte conterrà la domanda e la seconda la soluzione. Alcune osservazioni:

1. La funzione richiede come argomento una stringa contenente il nome del file senza estensione.

2. Nella prima riga: al nome del file viene aggiunta l'estensione: `nomefile+".dat"`, poi viene creato un oggetto file associato al nome `file_dati`.
3. Vengono lette tutte le sue linee in una lista di stringhe: `dati = file_dati.readlines()`.
4. L'ultima riga serve per vedere cosa abbiamo effettivamente letto.

```
    def gioco02(nomefile):
        file_dati = file(nomefile+".dat")
        dati = file_dati.readlines()
        print dati[0]
        for linea in dati[1:]:
            ese = linea.split("\t")
            domanda = ese[0]
            soluzione = ese[1][:-1]
            print "%s --> %s" % (domanda, soluzione)
```

Osservate che la seconda parte della riga contiene anche il carattere di a capo, lo eliminiamo, prima di associarlo alla variabile soluzione con l'istruzione: `soluzione=ese[1][:-1]`. Conviene provare il funzionamento inserendo un bel `print...` Ora aggiungiamo le variabili che permettano di conteggiare le risposte giuste e sbagliate e dentro nel ciclo il codice che ponga la domanda e memorizzi la risposta, che faccia il confronto tra la risposta e la soluzione. Alla fine del ciclo ci facciamo dire quante risposte giuste e sbagliate abbiamo dato e facciamo scrivere questa informazione in un file con lo stesso nome dei dati, ma con un'altra estensione:

```
def gioco03(nomefile):
    file_dati = file(nomefile+".dat")
    dati = file_dati.readlines()
    giuste = 0
    sbagliate = 0
    print dati[0]
    for linea in dati[1:]:
        ese = linea.split("\t")
        domanda = ese[0]
        soluzione = ese[1][:-1]
        risposta = raw_input(domanda+" -> ")
        if risposta == soluzione:
            print "Giusto!"
            giuste += 1
        else:
            print "Sbagliato, la soluzione è:", soluzione
            sbagliate += 1
    tot = giuste+sbagliate
    risultati = "Hai dato %s risposte giuste su %s (%s%%)" \
            % (giuste, tot, float(giuste)/tot*100)
    print risultati
    file(nomefile+".res", "w").write(risultati)
```

Alcune osservazioni:

1. Per scrivere in un file, bisogna aprirlo in scrittura, aggiungendo, l'argomento "w".

2. Nell'ultima riga, viene creato un oggetto file e viene scritta la stringa al volo senza usare variabili.
3. Se tento di aprire in lettura un file inesistente, ottengo un errore.
4. Se apro in scrittura un file inesistente, viene creato. Se esiste già, viene cancellato il suo contenuto.

Riassumendo

- Il comando file permette di aprire, in lettura o in scrittura un file, la sintassi è:

```
file(<nomefile>[, "w"])
```

- Per leggere tutto un file di testo si può usare il metodo `<file>.readlines()` e per scrivere una stringa in un file di testo `<file>.write(<stringa>)`.

Prova tu

1. Crea altri esercizi.
2. Scrivi un programma che legga un file di testo e ne scriva un altro con le righe numerate.
3. Scrivi un programma che legga un file e ne scriva un altro crittografato.

1.20 Dizionari

Come collegare valori agli oggetti immutabili.

Python mette a disposizione del programmatore un'altra potente struttura di dati: i dizionari. Assomigliano alle liste, ma gli indici, invece di essere numeri successivi sono oggetti immutabili qualunque, possono essere numeri, stringhe o tuple. Esempio:

```
>>> d={}                    # 1
>>> d[1]="pippo"            # 2
>>> d["gigi"]="pluto"       # 3
>>> d[(2,6)]="paperino"     # 4
>>> print d
{1: 'pippo', (2, 6): 'paperino', 'gigi': 'pluto'}
```

Alcune osservazioni:

1. `d` viene associato a un dizionario vuoto.
2. al numero `1` viene associato il valore "pippo"
3. alla stringa "gigi" viene associato il valore "pluto"
4. alla tupla (2, 6) viene associato il valore "paperino"
5. I dizionari mantengono le associazioni chiave-valore, non l'ordine degli elementi.

È possibile ottenere il valore associato ad una certa chiave o modificarlo usando la sintassi dell'indicizzazione:

```
>>> print d["gigi"]
pluto
>>> d["gigi"]="clarabella"
>>> print d["gigi"]
clarabella
```

Si può ottenere la lista di tutte le chiavi di un dizionario con il metodo `<dict>.keys()`:

```
>>> print d.keys()
[1, (2, 6), 'gigi']
```

Se cerchiamo di ottenere il valore collegato ad una chiave non esistente nel dizionario si ottiene un errore:

```
>>> print d["mario"]

[...]KeyError: 'mario'
```

La classe `dict` mette a disposizione due metodi che permettono di trattare il caso di chiavi non presenti nel dizionario. Il metodo `<dict>.get(<chiave>[, <valore>])` restituisce il valore collegato alla chiave se questa esiste, altrimenti restituisce il valore di default, non modifica il dizionario:

```
>>> print d.get("gigi", "clarabella")
pluto
>>> print d.get("mario", "clarabella")
clarabella
>>> d
{1: 'pippo', (2, 6): 'paperino', 'gigi': 'pluto'}
```

Il metodo `<dict>.setdefault(<chiave>[, <valore>])` restituisce il valore collegato ad una certa chiave, ma, se non esiste, crea un elemento con quella chiave e il valore passato come secondo argomento:

```
>>> print d.setdefault(1, "clarabella")
pippo
>>> d
{1: 'pippo', (2, 6): 'paperino', 'gigi': 'pluto'}
>>> print d.setdefault(2, "clarabella")
clarabella
>>> d
{1: 'pippo', 2: 'clarabella', (2, 6): 'paperino', 'gigi': 'pluto'}
```

Se vogliamo contare tutte le parole presenti in un certo testo (io ho scaricato da Internet `bibbia.txt`), possiamo:

1. predisporre un dizionario vuoto,
2. per ogni linea del testo:
   (a) spezzare la linea in una lista di parole e per ogni parola:
   (b) aumentare di 1 il contatore di quella parola (partendo dal valore 0 se non esiste).

3. estrarre tutte le chiavi e metterle in ordine, così è più comodo ricercare una parola,
4. stampare le coppie parola, numero di occorrenze.

L'algoritmo si traduce facilmente in Python:

```
def contaparole(nomefile):
    contatore = {}
    for linea in file(nomefile):
        for parola in linea.split():
            contatore[parola]=1+contatore.get(parola, 0)
    parole = contatore.keys()
    parole.sort()
    for parola in parole:
        print "%25s: %s" % (parola, contatore[parola])
```

Per contare le occorrenze delle parole della bibbia impiega una frazione di secondo, poi impiega qualche minuto per stamparle tutte.

Riassumendo

- Un dizionario vuoto è rappresentato da una coppia di parentesi graffe.
- Un dizionario è una contenitore che associa oggetti (valori) a oggetti immutabili (chiavi).
- Nei dizionari non è mantenuto l'ordine degli elementi, ma solo l'associazione chiave-valore.
- È possibile ottenere la lista delle chiavi di un dizionario con il metodo `<dict>keys()`.
- Si può accedere ad un valore usando l'indice del dizionario: `<dict>.[<chiave>]`.
- Si può creare un elemento assegnando un valore a una chiave: `<dict>.[<chiave>]=<valore>`.
- I due metodi:

```
<dict>.get(<chiave>[,<valore>]),
<dict>.setdefault(<chiave>[,<valore>]),
```

  permettono di trattare in modo uniforme i casi in cui non sappiamo se esiste un elemento con una certa chiave.

Prova tu

1. Modifica `contaparole()` in modo che stampi gli elementi dal più frequente al più raro.
2. Inserisci un "filtro" in modo che la punteggiatura e le maiuscole non abbiano influenza nei conteggi.

1.21 La programmazione orientata agli oggetti

Come definire e usare una nuova classe di oggetti.

Parlando dei dati Python ho spesso usato il termine "oggetto". Il termine non era casuale, tutto in Python è un *oggetto*. Gli oggetti sono delle entità formate da dati e algoritmi strettamente collegati tra di loro. I dati dell'oggetto, che rappresentano il suo stato, sono detti *attributi*, gli algoritmi che permettono di far funzionare l'oggetto sono detti *metodi*. In generale chi usa un oggetto non deve sapere come è organizzato al suo interno, quali sono i suoi attributi e quali algoritmi lo fanno funzionare, ma deve solo conoscere la sua interfaccia, cioè i metodi che permettono all'oggetto di interagire con gli altri oggetti. Gli oggetti sono elementi di una classe: una stringa è un oggetto della classe `str`, una particolare lista è un oggetto della classe `list`, ...

Nella programmazione orientata agli oggetti, una parte del lavoro consiste nel definire le classi. Una caratteristica importante delle classi è quella di ereditare da altre classi attributi e metodi. In questo modo si possono avere intere famiglie di classi imparentate tra di loro.

Come esempio costruiremo la classe delle frazioni facendola derivare dalla classe madre di tutte le classi: `object`. Il primo metodo che forniremo alla classe sarà il metodo `__init__(...)` che viene chiamato quando si crea un nuovo oggetto di questa classe:

```
class Frazione(object):

  def __init__(self, num, den):
    self.num=num
    self.den=den
```

Alcune osservazioni:

1. La prima riga dichiara che stiamo definendo una classe che si chiama Frazione e che discende dalla classe `object`.
2. Tutta la definizione della classe deve essere indentata.
3. Il metodo `__init__` ha un primo parametro convenzionalmente indicato con `self` che contiene il riferimento al particolare oggetto. Poi altri due parametri i cui valori vengono associati agli attributi `self.num` e `self.den`.

Creiamo il nostro primo oggetto della classe Frazione:

```
>>> f1=Frazione(4, 6)
>>> print f1
<__main__.Frazione object at 0xb6e00bec>
```

Funziona, ma non ci dice molto... Aggiungiamo un altro metodo con un nome convenzionale: il metodo `__str__` verrà chiamato tutte le volte ci sarà la necessità di convertire Frazione in stringa:

```
  def __str__(self):
    return "%s/%s" % (self.num, self.den)

>>> f1=Frazione(4, 6)
>>> print f1
4/6
```

Il risultato è già più carino! La prima operazione che ci hanno insegnato a eseguire con le frazioni è la riduzione ai minimi termini. Per farlo abbiamo però bisogno di una funzione che,

dati due numeri restituisca il massimo comun divisore (abbiamo già incontrato un'implementazione di questa funzione) e modifichiamo anche il metodo `__init__` in modo che dopo aver definito il numeratore e il denominatore riduca ai minimi termini la frazione:

```
  def __init__(self, num, den):
    self.num=num
    self.den=den
    self.riduci()

  def riduci(self):
    def macodi(a, b):
      while a<>b:
        if a>b: a-=b
        else:   b-=a
      return a
    d=macodi(self.num, self.den)
    self.num/=d
    self.den/=d

>>> f1=Frazione(4, 6)
>>> print f1
2/3
```

Aggiungiamo i metodi per addizionare e moltiplicare le frazioni:

```
  def __add__(self, altra):
    return Frazione(self.num*altra.den+self.den*altra.num,
                    self.den*altra.den)

  def __mul__(self, altra):
    return Frazione(self.num*altra.num,
                    self.den*altra.den)

>>> f1=Frazione(3, 6)
>>> f2=Frazione(4, 10)
>>> print "%s + %s = %s;  %s * %s = %s" % (f1, f2, f1+f2, f1, f2, f1*f2)
1/2 + 2/5 = 9/10;  1/2 * 2/5 = 1/5
```

Alcune osservazioni:

1. All'interno del metodo riduci è definita la funzione `macodi(a, b)`, effettivamente serve solo per ridurre ai minimi termini e quindi non è necessario che sia vista dall'esterno.
2. I metodi `__add__` e `__mul__` non hanno bisogno di ridurre ai minimi termini perché questo viene fatto al momento della creazione di una nuova frazione.
3. I metodi `__add__` e `__mul__` vengono chiamati quando in un'espressione deve essere eseguita l'addizione o la moltiplicazione tra due oggetti di questa classe.

Riassumendo

- L'istruzione `class` permette di creare una nuova classe.

- Una classe è formata dall'unione di dati, gli attributi e algoritmi, i metodi.
- Un metodo particolare è `__init__` che viene eseguito quando viene costruito l'oggetto.
- Il metodo `__str__` viene eseguito quando l'oggetto deve essere convertito in stringa.

Prova tu

1. La classe precedente non funziona se uno dei termini è 0, risolvi questi casi.
2. Aggiungi i metodi per eseguire sottrazioni e divisioni.

CAPITOLO 2

Numeri con Python

2. Numeri In questa sezione sono presentati dei problemi significativi da risolvere con Python puro o con l'aggiunta della libreria math. Per affrontare i problemi seguenti bisogna conoscere alcuni argomenti presentati nella sezione precedente.

2.1. Operazioni Cosa può fare un calcolatore che sa solo aggiungere o togliere uno

Prerequisiti funzioni con parametri, funzioni che restituiscono un valore, struttura di iterazione: while, Argomenti trattati concetti primitivi dell'aritmetica, algoritmi, grafi e linguaggio cicli. Problema Secondo il lavoro di Peano tutta l'aritmetica può essere derivata da 3 idee primitive: 1. zero 2. numero (naturale) 3. successore e da 5 proposizioni primitive: 1. Zero è un numero. 2. Il successore di ogni numero è un numero. 3. Due numeri non possono avere lo stesso successore. 4. Zero non è il successore di alcun numero. 5. Se lo zero ha una proprietà e per ogni numero che abbia quella proprietà anche il suo successore la ha allora tutti i numeri hanno quella proprietà. Tutta l'aritmetica può quindi essere derivata da questi tre concetti primitivi e da queste poche proposizioni. In particolare la quinta proposizione è detta anche principio di induzione e sta alla base delle definizioni ricorsive. Ispirati da Peano, ci poniamo il problema di stabilire quali siano le operazioni minime che un calcolatore deve saper fare per implementare l'aritmetica dei numeri naturali. Soluzione La soluzione non è unica, possiamo partire da un esecutore che sappia: 1. leggere dati, 2. confrontare un numero con 0, 3. aggiungere 1 a un numero, 4. togliere 1 a un numero, 5. ripetere delle istruzioni fin quando è vera una certa condizione, 6. restituire un risultato, 7. (per la divisione) eseguire porzioni diverse di codice a seconda del risultato di una condizione. Usando il linguaggio Python: 1. Per leggere i dati possiamo usare il meccanismo del passaggio di argomenti ad una funzione dotata di parametri. 2. Per confrontare il numero n con 0 possiamo usare le espressioni: n>0, n>=0 o n==0. 3. Per aggiungere 1 a un numero contenuto nella variabile n possiamo usare a piacere l'istruzione: n=n+1 oppure la più sintetica n+=1, del tutto equivalenti. 4. In modo analogo per togliere 1 possiamo scrivere n=n-1 o n-=1. 5. Per ripetere dei comandi possiamo usare l'istruzione while <condizione>:

<blocco di istruzioni>.

6. Per restituire un risultato usiamo l'istruzione return.

7. Per eseguire diverse porzioni di codice: if <condizione>:

<blocco di istruzioni>

else: <blocco di istruzioni>

Scriviamo delle funzioni che realizzino le quattro operazioni e il confronto di numeri. Iniziamo dall'addizione. Per addizionare due numeri contenuti in n1 e n2 posso trovare il successore di n1 tante volte quanto è contenuto in n2. Possiamo descrivere così l'algoritmo: definisci la somma di n1 e n2 così:

> **finquando n2 è maggiore di 0:** aumenta di 1 n1 e diminuisci di 1 n2
>
> alla fine, il risultato è contenuto in n1

La traduzione in Python non dovrebbe presentare grandi difficoltà. Aggiungiamo in fondo al file di programma una procedura test e un comando che viene eseguito quando viene eseguito il file come programma: def test():

> **t=[(4, 5), (5, 4), (367, 1), (1, 754), (0, 25), (36, 34),** (56, 0), (20, 4), (4, 20), (243, 243), (0, 0), (1, 1)]
>
> **for a, b in t:** print "%s+%s=%s" % (a, b, somma(a, b))

if __name__=="__main__": test()

Per realizzare la funzione prodotto, usando il metodo delle addizioni ripetute, avremo bisogno di una variabile locale che contenga le somme parziali. La descrizione dell'algoritmo potrebbe essere: definisci il prodotto di n1 e n2 così:

> in s metti zero fin quando n2 è maggiore di 0:
>
> > in s metti la somma di s e n1 diminuisci di 1 n2
>
> alla fine, il risultato è contenuto in s

Per implementare correttamente la funzione quoziente, dobbiamo usare un'altra istruzione Python: l'istruzione if. Infatti non sempre è possibile eseguire le divisioni tra numeri naturali, non è definita una divisione con il divisore uguale a zero. Quindi la funzione quoziente, prima di procedere deve controllare il valore del secondo numero e terminare con un messaggio di errore se è uguale a zero.

Altra caratteristica importante dei numeri è il fatto che possono tutti essere confrontati tra di loro. Usando le solite istruzioni possiamo realizzare le tre funzioni di confronto: maggiore, minore, uguale. Ci sono diversi modi per realizzarle, uno è questo: definisci maggiore n1 di n2 così:

> **fin quando n2 è maggiore di 0:** diminuisco sia n1 sia n2 di 1
>
> alla fine, il risultato è equivalente a in n1>0

Proviamo ad affrontare il problema con un approccio diverso, chiudiamo questo file e apriamone uno nuovo. Se il nostro esecutore è in grado di gestire funzioni con parametri che restituiscono valori, possiamo fare a meno dell'istruzione while e usare la ricorsione. Abbiamo già osservato come il principio di induzione sia vicino al meccanismo della ricorsione, ora vedremo come scrivere delle definizioni ricorsive e tradurle in funzioni. Una definizione ricorsiva definisce qualcosa nei termini della cosa stessa. Non sempre le definizioni ricorsive funzionano, perché siano accettabili bisogna che trasformino un caso in un caso più semplice e abbiano una condizione di terminazione. Ad esempio: addizionare i due numeri 6 e 3 è equivalente ad addizionare 7 e 2 questo è equivalente ad addizionare 8 a 1 e questo è equivalente ad addizionare 9 a 0. Si può osservare che il secondo addendo diventa sempre più piccolo, il problema

diventa più semplice. La condizione di terminazione in questo caso è: la somma di un numero con 0 è il numero stesso. nel caso generale possiamo descrivere la definizione in questo modo:

è n1 se n2=0

somma di n1 e n2 |

altrimenti è somma di (n1+1) con (n2-1)

La traduzione in Python è pressoché letterale: def somma(n1, n2):

“”“somma(n1, n2) -> restituisce la somma n1+n2.”“” if n2==0: return n1 return somma(n1+1, n2-1)

Possiamo osservare che le due righe della definizione si sono tradotte in due righe della funzione. La prima riga della funzione è un commento utile come documentazione. Scrivi anche la procedura di test e controlla il funzionamento corretto della funzione. Se provi questa funzione con numeri molto grandi, Python termina con un errore perché non è in grado di gestire troppe funzioni ricorsive annidate una all’interno dell’altra.

Riassumendo Un Calcolatore può implementare tutta l’aritmetica partendo da poche semplici operazioni. I concetti di successore, predecessore, confronto con zero e iterazione permettono di realizzare l’addizione, la sottrazione, la moltiplicazione e la potenza. Per realizzare la divisione dobbiamo anche controllare che il divisore sia diverso da zero. Le stesse semplici operazioni permettono di realizzare anche il confronto tra numeri. Le operazioni e il confronto possono anche essere definiti in modo ricorsivo. Le definizioni ricorsive risultano molto sintetiche, ed espressive. È facile tradurre definizioni ricorsive in funzioni equivalenti. Il linguaggio Python non tratta le funzioni ricorsive in modo molto efficiente, altri linguaggi lo fanno molto meglio.

2.2. Array di interi Cosa fare con vettori di numeri interi. Prerequisiti funzioni con parametri, funzioni che restituiscono un valore, struttura di selezione: if, struttura di iterazione: for, Sequenze di dati: liste.

Argomenti trattati Lettura di numeri da tastiera, inizializzazione di una lista con valori casuali, somma di una sequenza di numeri, ricerca del minimo, verifica della presenza di un elemento in una sequenza, ordinamento di una sequenza.

Problema Leggere numeri e inserirli in una lista, visualizzare i numeri contenuti in una lista, svolgere alcune operazioni di base su una lista di interi: somma, prodotto, ricerca del minimo, ricerca del del massimo, ordinamento. Soluzione Una struttura di dati importante in informatica è costituita dagli array. Gli array sono sequenze indicizzabili di elementi. In Python gli array possono essere emulati dall’oggetto list e lo si può costruire racchiudendo tra parentesi quadre una sequenza di oggetti separati da virgole: >>> a=[1, 3, 5, 7, 9, 11] Dopo questo comando, l’oggetto a contiene i primi 6 numeri dispari. Si può ottenere l’ennesimo elemento di una lista scrivendo il nome della lista seguito dall’indice posto tra parentesi quadre. In Python, il primo elemento di ogni lista ha sempre indice 0. >>> print a[0] 1 >>> print a[3] 7 Ci proponiamo di scrivere alcune funzioni di base per l’elaborazione delle liste di numeri. Per incominciare ci serve una funzione che restituisca un array, di numeri interi, letto da tastiera. La funzione deve continuare a leggere numeri fnquando viene introdotto un valore particolare detto sentinella. In metalinguaggio: per leggilistainteri(sentinella) fa:

prepara una lista vuota avvia un ciclo infinito che:

leggi un valore se è uguale alla sentinella: termina il ciclo altrimenti: aggiungi il valore alla lista

restituisci la lista.

Possiamo partire: menu File - New Window, solite intestazioni, menu File - Save. Scriviamo questa prima funzione: import random

def leggilistainteri(sentinella="0"): result=[] while True:

e=raw_input("introduci un numero (%s per finire): " % sentinella)

if e==sentinella: break else: result.append(int(e))

return result

Alcune osservazioni: 1. Il comando import random servirà per la prossima funzione, ma io preferisco metterlo all'inizio del programma. 2. Il parametro sentinella ha come valore di default "0". ciò vuol dire che le due chiamate di funzione seguenti sono equivalenti: leggilistainteri() leggilistainteri("0") 3. Quando Python incotra il comando break esce dal ciclo, questo permette di eseguire un ciclo while True: che altrimenti sarebbe infinito.

Per le prove seguenti sarà utile avere una funzione che restituisca una lista di enne valori casuali. per listacasuale(n, mi=0, ma=100) fa:

prepara una lista vuota per n volte:

aggiungi alla lista un numero casuale tra mi e ma

restituisci la lista.

Questa volta, poiché il numero di elementi è già noto prima di avviare il ciclo, conviene usare un ciclo for: def listacasuale(n, mi=0, ma=100):

result=[] for i in xrange(n):

result.append(random.randrange(mi, ma))

return result

prima di procedere conviene verificare le due funzioni. <F5> per far valutare a Python il nostro programma, poi nell'ambiente IDLE proviamo le due nuove funzioni. Nel mio caso, dopo la correzione di un po' di errori, hanno mostrato il seguente funzionamento: >>> a=leggilistainteri() introduci un numero (0 per finire): 34 introduci un numero (0 per finire): 65 introduci un numero (0 per finire): 2 introduci un numero (0 per finire): 0 >>> a [34, 65, 2] >>> b=listacasuale(10) >>> b [20, 1, 88, 77, 25, 62, 43, 60, 20, 42] >>> b=listacasuale(10, 0, 1000) >>> b [117, 266, 960, 567, 39, 122, 581, 289, 759, 73] Bene, messi a punto questi due strumenti, passiamo a risolvere alcuni tipici problemi sui vettori di numeri. Il primo consiste nel trovare il minimo valore contenuto nell'array. Il trucco consiste nel far finta che il minimo sia il primo elemento e, poi, confrontare questo valore con gli altri cambiandolo se se ne trova uno più piccolo. In metalinguaggio: per minimo(lista) fa:

metti in minimo il primo elemento della lista per ogni elemento della lista:

se l'elemento è più piccolo del minimo metti in minimo l'elemento

restituisci il minimo.

La traduzione risulta abbastanza semplice: ricordiamoci di mettere una stringa di commento come prima riga della funzione: def minimo(lista):

“”“Restituisce l’elemento minimo di una lista.”“” mi=lista[0] for e in lista:

if e<mi: mi=e

return mi

In qualche (raro) caso può interessare non l’elemento minimo, ma la posizione dell’elemento minimo. La funzione in questo caso è un po’ più complicata perché dobbiamo confrontare gli elementi della lista, ma dobbiamo memorizzare e restituire un indice. In metalinguaggio: per indiceminimo(lista) fa:

metti in minimo il primo indice della lista (0) per ogni indice della lista:

se (l’elemento con questo indice è più piccolo

dell’elemento che ha per indice minimo) allora:

metti in minimo questo indice

restituisci il minimo.

Accidenti, è più difficile da scrivere in metalinguaggio che in Python: def indiceminimo(lista):

“”“Restituisce l’indice dell’elemento min. di lista.”“” indmin=0 for indice in range(len(lista)):

if lista[indice]<lista[indmin]: indmin=indice

return indmin

Un’altra funzione utile è quella che calcola la somma degli elementi di una lista: per somma(lista) fa:

poni a 0 la somma parziale per ogni elemento della lista:

aggiungi il suo valore alla somma parziale

restituisci la somma parziale che, ora, è il totale.

La traduzione in Python non presenta sorprese: def somma(lista):

“”“Restituisce la somma degli elementi di una lista.”“” somma=0 for elemento in lista:

somma+=elemento

return somma

A volte è utile una funzione che controlla se un certo elemento è contenuto oppure no in un array, oppure che ci restituisce l’indice di un determinato elemento. Due funzioni che risolvono questi problemi possono essere scritte così: def appartiene(elemento, lista):

“”“Restituisce True se elemento appartiene a lista.”“” for e in lista:

if e==elemento: return True

return False

def indicedi(elemento, lista):

“”“Restituisce l’indice della prima occorrenza di elemento in lista, retituisce -1 se l’elemento non è presente nella lista.”“”

for ind, ele in enumerate(lista): if ele==elemento: return ind

return -1

Alcune osservazioni: 1. Il comando return termina il ciclo, la funzione e restituisce come risultato il valore che lo segue. 2. Poiché gli indici degli array iniziano da 0, per indicare che un elemento non è presente nell’array possiamo usare un numero negativo. 3. La funzione enumerate restituisce le coppie formate da un indice e il corrispondente elemento. 4. La funzione indicedi restituisce l’indice del primo elemento uguale a quello cercato presente nell’array.

Per provare le funzioni appena aggiunte, possiamo creare una lista e chiamarle con diversi argomenti: >>> a=[3,6,2,8,6,4,1] >>> print appartiene(6, a) True >>> print appartiene(23, a) False >>> print indicedi(4, a) 5 >>> print indicedi(6, a) 1 Un altro importante problema relativo agli array è quello dell’ordinamento. si parte con un array disordinato e si vuole ottenere uno ordinato. È un problema importante, perché è facile scrivere algoritmi poco efficienti, ma quelli efficienti risultano piuttosto complicati. Nei problemi pratici è facile dover ordinare array di migliaia o milioni di elementi e quindi non è secondario avere a disposizione un algoritmo più efficiente. Possiamo realizzare una prima soluzione nella quale la funzione cerca l’elemento più piccolo della lista, lo toglie e lo mette in fondo ad una nuova lista, procedendo così fin quando la prima lista è vuota. Per questa funzione useremo un altro metodo delle liste, metodo che restituisce un elemento togliendolo dalla lista: <lista>.pop(<indice>): per insertionsort(lista) fa:

il risultato è la lista vuota finquando la lista di partenza non è vuota:

trova l’indice dell’elemento minimo togli questo ele. dalla lista e mettilo nel risulato

restituisci il risultato

In Python: def insertionsort(lista):

“”“Rest. una lista con gli ele. di lista ordinati.”“” elementi=lista[:] result=[] while elementi!=[]:

im=indiceminimo(elementi) result.append(elementi.pop(im))

return result

Il bubblesort confronta ogni elemento con tutti quelli che lo seguono scambiando, gli elementi quando non sono in ordine. Questa funzione non produce un nuovo array disfando quello dato come argomento, ma ordina direttamente l’array passato come argomento. def bubblesort(lista):

“”“Ordina “sul posto” una lista.”“” for i in xrange(len(lista)-1):

for j in xrange(i+1, len(lista)):

if lista[j]<lista[i]: lista[j], lista[i] = lista[i], lista[j]

Riassumendo Quando si devono elaborare degli array spesso si usa il ciclo for applicato direttamente agli elementi dell'array. Le funzioni che abbiamo scritto forniscono degli strumenti di base utilissimi in tutti i programmi che operano con array. Sono così utili che Python le fa già tutte senza bisogno di programmarle... il comando help(list) ci mostra tutto quello che le liste sanno già fare senza bisogno del nostro intervento. Come dicono gli hacker "due settimane di sperimentazione in laboratorio possono far risparmiare due buone ore di studio dei manuali in biblioteca"...

2.3. Criteri di divisibilità Dove si insegna a Python a utilizzare i criteri di divisibilità. Prerequisiti funzioni con parametri, struttura di selezione: if ..else, struttura di iterazione: while, dati: numeri, stringhe e liste.

Argomenti trattati i criteri di divisibilità, conversioni da numero a stringa e viceversa, costruzione di liste, ricorsione.

Problema Scrivere una libreria di funzioni che implementi i criteri di divisibilità dei numeri naturali. La funzione div2(n) deve verificare che il numero n sia divisibile per 2 e restituire True se n è pari e False se non lo è. Devo poter dare i seguenti comandi ottenendo le risposte corrette: >>> print div2(5) False >>> print div2(613264) True >>> print div5(64343) False ...

Soluzione Ovviamente per prima cosa bisogna conoscere i criteri di divisibilità, ne riporto, di seguito una possibile definizione, i libri li spiegano molto meglio... Divisibilità per 2: i seguenti numeri sono divisibili per 2: 0, 2, 4, 6, 8, un numero è divisibile per 2 se la sua ultima cifra è un numero divisibile per 2. Divisibilità per 4: i seguenti numeri sono divisibili per 4: 0, 4, 8, 12, 16, un numero è divisibile per 4 se lo è anche il numero che si ottiene togliendo 20 un numero è divisibile per 4 se le sue ultime 2 cifre formano un numero divisibile per 4. Divisibilità per 8: i seguenti numeri sono divisibili per 8: 0, 8, 16, 24, 32 un numero è divisibile per 8 se lo è anche il numero che si ottiene togliendo 40 un numero è divisibile per 8 se le sue ultime 3 cifre formano un numero divisibile per 8. Divisibilità per 5: i seguenti numeri sono divisibili per 5: 0, 5, un numero è divisibile per 5 se la sua ultima cifra è un numero divisibile per 5. Divisibilità per 25: i seguenti numeri sono divisibili per 25: 0, 25, 50, 75, un numero è divisibile per 25 se le sue ultime 2 cifre formano un numero divisibile per 25. Divisibilità per 10: un numero è divisibile per 10 se la sua ultima cifra è 0. Divisibilità per 100: un numero è divisibile per 100 se le sue ultime 2 cifre formano il numero 0. Divisibilità per 3: i seguenti numeri sono divisibili per 3: 0, 3, 6, 9, un numero è divisibile per 3 se la somma delle sue cifre è un numero divisibile per 3. Divisibilità per 9: i seguenti numeri sono divisibili per 9: 0, 9, un numero è divisibile per 9 se la somma delle sue cifre è un numero divisibile per 9. Divisibilità per 11: i seguenti numeri sono divisibili per 11: 0, 11, un numero è divisibile per 11 se la differenza della somma delle sue cifre di posto pari e la somma delle sue cifre di posto dispari è un numero divisibile per 11. Divisibilità per 7: i seguenti numeri sono divisibili per 7: 0, 7, un numero è divisibile per 7 se la differenza tra doppio dell'ultima cifra e il numero formato dalle altre cifre è un numero divisibile per 7. Tutte le definizioni date sopra sono ricorsive, ma le funzioni che faremo non utilizzeranno questa tecnica. Il criterio più semplice è quello della divisibilità per 2, è semplice, ma richiede di poter estrarre da un numero, il numero formato dalla sua ultima cifra. Quindi, prima di affrontare il criterio, dobbiamo scrivere una funzione che dato un numero restituisca il numero formato dalla sua ultima cifra, cioè che funzioni così: >>> print destra1(743843) 3 Un metodo, può essere questo: trasformo un numero in stringa con il comando str(<numero>) prendo il carattere più a destra <stringa>[-1] lo trasformo in numero con il comando int(<stringa>) restituisco questo numero con il comando return <valore> La funzione potrebbe essere: def destra1(numero):

“”“destra1(numero) -> ultima cifra del numero.”“” numero_in_stringa=str(numero) ultimo_carattere=numero_in_stringa[-1] risultato=int(ultimo_carattere) return risultato

Alcune osservazioni: 1. la funzione destra1 ha un parametro che si chiama numero e che conterrà il numero da cui trarre l'ultima cifra. 2. La prima riga dopo l'intestazione della funzione è un commento che dovrebbe ricordare a noi, o spiegare a chi usa la funzione, qual è l'effetto di questo pezzo di codice. 3. Ho usato parecchie variabili cercando di dare loro dei nomi abbastanza espliciti. 4. Ogni funzione usata modifica il valore assegnato all'ultima variabile.

L'ultima osservazione permette di riscrivere il pezzo di codice precedente usando il paradigma funzionale, qualcosa del genere: Restituisci - la trasformazione in intero - dell'ultimo carattere - della stringa ottenuta dal numero. return int(str(numero)[-1]) La funzione diventa quindi: def destra1(numero):

“”“destra1(numero) -> ultima cifra del numero.”“” return int(str(numero)[-1])

Dopo aver eseguito il file che contiene la funzione (tasto <F5>), nell'ambiente IDLE si può provarla con diversi valori: >>> print destra1(65326432) 2 >>> print destra1(8676) 6 ... Ora abbiamo lo strumento per risolvere il criterio di divisibilità per 2 (e quali altri?). La funzione dovrà implementare questa logica: se l'ultima cifra del numero è 0 o 2 o 4 o 6 o 8 restituisci True altrimenti restituisci False La funzione potrebbe quindi essere: def div2(n):

“”“n e' divisibile per 2.”“” if (destra1(n)==0 or destra1(n)==2 or

destra1(n)==4 or destra1(n)==6 or destra1(n)==8):

return True

else: return False

Alcune osservazioni: 1. La funzione rispecchia letteralmente quanto progettato. 2. La funzione destra1(n) viene chiamata 5 volte per dare sempre lo stesso risultato, questo non è efficiente! 3. Si può fare a meno dell'istruzione if.

Per ovviare al problema del punto 2 si potrebbe usare una variabile e chiamare la funzione destra1(n) una sola volta, migliorerebbe l'efficienza. Ma Python ci mette a disposizione un altro strumento che rende il codice ancora più leggibile: destra1(n) in (0, 2, 4, 6, 8) Che verifica se l'ultima cifra di n è contenuta nella “tupla” (0, 2, 4, 6, 8). E dato che questa espressione dà come risultato un valore vero o falso, non abbiamo bisogno di usare l'istruzione if ma basta che la funzione restituisca esattamente questo valore. La funzione diventa quindi: def div2(n):

“”“n e' divisibile per 2.”“” return destra1(n) in (0, 2, 4, 6, 8)

Nota: la concisione e la leggibilità sono caratteristiche importanti nella programmazione! Per completare l'opera possiamo costruire una funzione “test” che metta alla prova div2. La possiamo fare in molti modi diversi. Io ho pensato di provare con 5 casi di numeri divisibili per 2 e 5 di numeri non divisibili per due: def test():

print “ndivisibilita' per 2” numeri=[0, 2, 8, 9456546, 10546454674670,

1, 7, 3163, 5*11*7*13, 7*15*19*33]

for a in numeri: print “a=%s; div2: %s” % (a, div2(a))

Alcune osservazioni: 1. I simboli ”n” all’interno di una stringa significano: “va a capo”. 2. Qui viene utilizzato un ciclo for, la sintassi è: for <variabile> in <sequenza>:

<blocco di istruzioni>

e significa all’incirca: per ogni elemento della sequenza esegui il blocco di codice. Nel nostro caso, per ogni numero contenuto in numeri stampa la stringa formattata. 3. Il comando print è seguito da tre oggetti: 1. una stringa che contiene i simboli: “%s” che sono dei segnaposti, 2. l’operatore di formattazione “%”, 3. una tupla che contiene tanti oggetti quanti sono i segnaposti. Prima di stampare la stringa, vengono calcolati gli oggetti presenti nella tupla e inseriti in ordine nella stringa dove ci sono i segnaposti.

Conviene aggiungere in fondo al file l’istruzione: if __name__ == “__main__”: test () In questo modo quando viene eseguito il programma vene eseguita automaticamente la funzione test(). Mentre se usiamo il programma come libreria di funzioni test() non viene eseguito.

Il criterio di divisibilità per 4 assomiglia molto a quello per 2, ma richiede che si estragga il numero composto dalle ultime due cifre, partiamo da qui. Visto come era semplice la funzione per estrarre una cifra possiamo modificarla per estrarne 2: def destra2(numero):

“”“destra2(numero) -> ultime due cifre del numero.”“” return int(str(numero)[-2:])

Funziona, se il numero ha più di 2 cifre ma se ne ha solo una? Qualche prova ci fa vedere che Python è abbastanza furbo e se gli si chiede di restituire più cifre di quelle contenute in una stringa restituisce l’intera stringa. Possiamo ora affrontare la divisibilità per 4. prima chiariamo il concetto: estraggo le ultime 2 cifre del numero, continuo a togliere 20 finché diventa più piccolo di 20, controllo che quello che ho ottenuto sia un multiplo di 4. In Python: def div4(n):

“”“n e’ divisibile per 4.”“” n2=destra2(n) while n2>=20: n2-=20 return n2 in [0, 4, 8, 12, 16]

Alcune osservazioni: 1. La sintassi del ciclo while è: while <condizione>:

<blocco istruzioni>

2. Se il ciclo while deve ripetere una sola istruzione allora è possibile scriverla sulla stessa riga dell’intestazione dopo il duepunti.
3. L’istruzione n2-=20 significa: togli 20 dal numero collegato alla parola n2 e collega il risultato ancora alla stessa parola. È equivalente a n2=n2-20, ma è un po’ più sintetico.

Il criterio di divisibilità per 3 è di tipo diverso, non coinvolge solo la parte finale del numero, ma tutto il numero: richiede che si faccia la somma di tutte le cifre. Python mette a disposizione una funzione che somma gli elementi di una lista, solo se questi sono numeri. Abbiamo quindi bisogno di una funzione che trasformi un numero in una lista di cifre. Lo si può fare in diversi modi, ma usiamo un metodo tipico di Python: la costruzione di liste. Non è facilissimo da capire, ma è sintetico ed elegante. def cifre(numero):

“”“cifre(numero) -> lista delle cifre del numero.”“” return list(str(numero))

Si potrebbe tradurre con: restituisci la lista che contiene la trasformazione in intero di ogni carattere della stringa equivalente al numero. Oppure: Prendi il numero, lo trasformi in stringa, per ogni carattere della stringa, lo trasformi in intero e lo aggiungi alla lista. A questo punto scrivere la funzione che implementa il criterio di divisibilità per 3 diventa una passeggiata: def div3(n):

“”“n e’ divisibile per 3.”“” n=sum(cifre(n)) return n in [0, 3, 6, 9]

Modifichiamo la funzione test in modo che verifichi anche la nuova funzione: def test():

print “ndivisibilita’ per 2” numeri=[0, 2, 8, 9456546, 10546454674670,

1, 7, 3163, 5*11*7*13, 7*15*19*33]

for a in numeri: print “a=%s; div2: %s” % (a, div2(a)) ... print “ndivisibilita’ per 3” numeri=[0, 3, 9, 24, 36576798,

1, 7, 3163, 25466737, 5*11*7*13,]

for a in numeri: print “a=%s; div3: %s” % (a, div3(a))

Proviamolo... peccato che non funzioni: il numero 36576798 non viene riconosciuto come un multiplo di 3! Come mai? La somma delle cifre dà un numero di due cifre, quindi bisogna sommarne ancora le cifre e ripeterlo finquando la somma ottenuta rimane maggiore di 9. Qui ci viene in soccorso il ciclo while: def div3(n):

“”“n e’ divisibile per 3.”“” while n>9: n=sum(cifre(n)) return n in [0, 3, 6, 9]

Il criterio di divisibilità per 11 richiede che sommiamo le cifre di posto pari e le cifre di posto dispari. Abbiamo già una funzione che, dato un numero restituisce la lista delle sue cifre, dobbiamo costruire una funzione che data una lista restituisce due liste contenenti gli elementi di posto pari e di posto dispari. Possiamo utilizzare un ciclo for con una particolare istruzione Python che restituisce non solo l’elemento, ma anche la sua posizione: enumerate(<sequenza>). La funzione deve: predisporre due liste vuote, p e d, scorrere la lista data e: se la posizione è pari: aggiungerla alla lista p, altrimenti: aggiungerla alla lista d, restituire le due liste. def separa(lista):

“”“Separa in due liste gli elementi di posto pari e di posto dispari.”“”

pari=[] dispari=[] for ind, cifra in enumerate(lista):

if div2(ind): p.append(cifra) else: d.append(cifra)

return pari, dispari

Alcune osservazioni: 1. Il metodo append aggiunge un elemento in coda ad una lista la sintassi è: <lista>.append(elemento). Può essere usato per costruire liste partendo da una lista vuota e aggiungendo man mano gli elementi. 2. A differenza di altri linguaggi, l’istruzione return può restituire anche più elementi. In questo caso, restituisce 2 liste. Chi chiama questa funzione deve caricare due variabili con il suo risultato, ad es.: postopari, postodispari = separa(c) Riassumendo Abbiamo visto come scrivere alcune funzioni non solo funzionanti, a anche efficienti, sintetiche e facilmente comprensibili. Abbiamo costruito assieme le seguenti funzioni di utilità: destra1(numero), destra2(numero), cifre(numero), separa(lista) e i seguenti criteri di divisibilità: div2(n), div4(n), div3(n). Hai costruito le seguenti funzioni che traducono le definizioni dei criteri di divisibilità: div5(n), div10(n), div25(n), div100(n), div1000(n), destran(numero, cifre), div8(n), div1000(n), div9(n).

2.4. Divisibilità Come trovare il massimo comune divisore e scoprire se un numero è perfetto. Prerequisiti operatori: % e ==. funzioni ricorsive, struttura di selezione: if ..else, struttura di iterazione: while e for, dati: numeri e liste.

Argomenti trattati massimo comune divisore, divisori di un numero, numeri perfetti.

Problema Scrivere una libreria funzione che ricerchi i numeri perfetti, i numeri cioè che sono uguali alla somma dei loro divisori tranne il numero stesso.

Soluzione Gli elaboratori elettronici permettono di affrontare e risolvere alcuni problemi legati alla divisibilità. Python mette a disposizione alcuni comandi per questo. Il primo di questi è l'operatore: "%". Lo stesso operatore ha significati completamente diversi a seconda del contesto in cui si trova. Serve per formattare stringhe: "somma=%s" % somma, ma tra due numeri % non viene utilizzato per qualcosa che riguarda le percentuali, ma serve per calcolare il resto di una divisione. Fin dalle elementari ci hanno insegnato che 13 diviso 3 fa 4 con il resto di 1. In Python il resto di questa divisione lo si trova con l'espressione: 13 % 3:

13 % 3 -> 1 15 % 4 -> 3 55 % 10 -> 5 457 % 40 ->17 ... Ma cosa centra il resto della divisione con la divisibilità? In matematica si dice che un numero è divisibile per un altro quando il resto della divisione è uguale a zero. Perciò si dice che 15 è divisibile per 5 perché il resto di 15 diviso 5 è 0. Tutti i numeri pari sono divisibili per 2 perché il resto della divisione tra un numero pari e 2 è proprio 0. L'operatore % che dà il resto di una divisione permette di determinare se un numero divide esattamente un altro. Possiamo scrivere una funzione che trova se un numero è divisibile per 3: def div3(num):

return num % 3 == 0

E così via, ad esempio per controllare se un numero è divisibile per 7 si può costruire la funzione: def div7(num):

return num % 7 == 0

Una generica funzione che controlla se il numero n è divisibile per il numero d, può essere: def divisibile(num, div):

return num % div == 0

In vari casi nei calcoli aritmetici è necessario trovare il minimo comune multiplo e il massimo comune d

è a se a=b

il mcd di a e b | altrimenti è il mcd tra

la differenza tra a e b e
il più piccolo tra a e b

Tradotto in Python con una funzione ricorsiva: def mcd1(a, b):

if a==b: return a elif a>b: return mcd1(a-b, b) else: return mcd1(b-a, a)

Vari altri algoritmi sono possibili, tra tutti spicca per semplicità ed eleganza il seguente, composto da tre righe di codice: def mcd(a, b):

''‘Massimo Comune Divisore tra a e b’‘’ while b:

a, b = b, a%b

return a

Possiamo scrivere ora una funzione che restituisca la lista dei divisori di un numero. Ci sono molti modi per farlo, un'idea può essere: predisporre una lista vuota, per d che va da 1 a

n: se d divide n aggiungi d alla lista restituisci la lista così ottenuta Tradotto in Python: def divisori0(num):

> ''‘Restituisce la lista dei divisori di num. Usa un ciclo for e append per popolare la lista.’‘ result=[] for div in range(1, num+1):
>
> if (num % div)==0: result.append(div)
>
> return result

La funzione precedente è formata da tre blocchi: l'inizializzazione della variabile che conterrà la lista dei divisori, il ciclo che popola questa lista, l'istruzione che termina la funzione restituendo come risultato la lista dei divisori. Python mette a disposizione una sintassi più compatta per popolare una lista. Lo stesso risultato della funzione precedente può essere ottenuto la una sola riga di istruzioni: def divisori1(num):

> ''‘Restituisce la lista dei divisori di n. Usa la costruzione di liste, list comprehension.’‘ return [div for div in range(1, num+1) if (num % div)==0]

Che può essere tradotto con: restituisci la lista formata dagli elementi d tali che d appartengono all'intervallo [1, num] e div è divisore di num. Le funzioni precedenti sono poco efficienti, è possibile migliorare la ricerca dei divisori di un numero in diversi modi: 1. Python permette di scrivere funzioni che restituiscono più oggetti, in particolare ha una funzione primitiva che restituisce quoziente intero e resto di una divisione tra interi. Nella shell di IDLE facciamo qualche prova: >>> print divmod(17, 5) 3, 2 2. Ogni volta che troviamo un divisore, possiamo conoscerne anche un'altro: se div è divisore anche num/div lo è. 3. Tra la metà di num e num stesso non ci sono divisori, quindi possiamo dimezzare l'intervallo di ricerca. 4. Anzi, se ogni volta che incontriamo un divisore div, inseriamo anche num/div, possiamo fermarci nella ricerca quando div supera num/div. Tenendo conto di queste osservazioni possiamo scrivere una funzione più complicata, ma moto più efficiente: def divisori(n):

> result=[] d=1 q=n r=0 while d<=q:
>
> **if r==0:** if d==q: result+=[d] else: result+=[d, q]
>
> d+=1 q, r = divmod(n, d)

 # print d, q, r result.sort() return result

Stampare i valori di alcune variabili all'interno di un ciclo, può aiutare a capirne il suo funzionamento o a scoprire degli errori. Quando non serve, la riga che produce la stampa può essere trasformata in un commento semplicemente anteponendo il carattere "#" al comando. Una volta che abbiamo i divisori di un numero possiamo sommarli tutti tranne il numero stesso. Ad esempio: I divisori di 3 sono: [1, 3], la somma dei divisori di 3 tranne 3 stesso è: 1. I divisori di 4 sono: [1, 2, 4], la somma dei divisori di 4 tranne 4 stesso è: 3. I divisori di 5 sono: [1, 5], la somma dei divisori di 5 tranne 5 stesso è: 1. I divisori di 6 sono: [1, 2, 3, 6], la somma dei divisori di 6 tranne 6 stesso è: 6. ... Ora i numeri che sono ugualia alla somma dei loro divisori, tranne il numero stesso, sono detti numeri perfetti. Python ha una funzione che restituisce la somma di tutti i numeri contenuti in una lista; possiamo usarla per calcolare la somma dei divisori. Usiamo IDLE per fare un po' di provacce. Ad esempio se voglio trovare la somma dei divisori di 4 o di 12 posso scrivere >>> print sum([1, 2, 4]) 7 >>> print sum([1, 2, 3, 4, 6, 12]) 28 O più semplicemente posso far calcolare a Python la lista dei divisori: >>> print sum(divisori(4)) 7 >>> print sum(divisori(12)) 28 Peccato che la somma che ci interessa

sia quella di tutti i divisori tranne l'ultimo. Python mette a disposizione un metodo per estrarre una sottolista di una lista, per estrarre una "fetta" di una lista. Per maggiori informazioni si può (ri)guardare il capitolo sulle Liste e Tuple. Spezzettiamo tutte le operazioni logiche che chiediamo a Python di compiere in modo da renderci conto di quello che succede: >>> # mettiamo in d4 la lista dei divisori di 4 >>> d4=divisori(4) >>> # per sicurezza stampiamo il contenuto di d4 >>> print d4 [1, 2, 4] >>> # bene, ora mettiamo in d4mu il contenuto di d4 meno l'ultimo elemento >>> d4mu=d4[:-1] >>> # ora stampiamo la somma degli elementi di d4mu >>> print sum(d4mu) 3 Ora possiamo mettere tutto assieme: stampiamo la somma della fetta dei divisori di 4: >>> print sum(divisori(4)[:-1]) 3 Bene possiamo ora creare una funzione che restituisca la "somma" dei divisori di un numero passato come argomento: def sommadiv(n):

return sum(divisori(n)[:-1])

E, infine, possiamo scrivere una funzione che restituisce "vero" o "falso" a seconda che un numero sia "perfetto" oppure no: def perfetto(n):

return n==sommadiv(n)

Possiamo usare la funzione precedente per stampare tutti i numeri perfetti minori di 100: def perfetti100():

for n in range(2, 101): if perfetto(n): print n, "è perfetto", divisori(n)

Oppure creare una procedura che vada avanti finché non viene premuto il tasto <Ctrl-c> e che continua a cercare numeri perfetti: def perfetti0():

n=2 while True:

if perfetto(n): print n, "è perfetto", divisori(n) n+=1

Un perfezionamento di questa procedura permette di partire da un numero a propria scelta, se non viene indicato è 2, e di andare avanti finché l'utente non decide di interrompere il ciclo. Quando il ciclo viene interrotto, premendo il tasto <Ctrl-c>, la procedura, prima di terminare, stampa il valore attuale di n. def perfetti(n=2):

while True:

try: if perfetto(n): print n, "è perfetto", divisori(n) n+=1

except: print "sono arrivato al numero", n return

Riassumendo L'operatore "%" restituisce il resto della divisione tra due interi. la funzione divmod(n, d) restituisce il quoziente e il reato della duivisione tra due numeri. Si possono scrivere funzioni che sfruttando, la selezione, l'iterazione o la ricorsione, esplorano proprietà legate alla divisibilità dei numeri naturali.

2.5. Numeri primi Come realizzare il crivello di Eratostene, Come cercare i numeri primi. Prerequisiti operatori: //, %, <, e ==. funzioni con parametri, struttura di selezione: if ..else, struttura di iterazione: while e istruzione break, liste. Argomenti trattati crivello di Eratostene, numeri primi, ottimizzazione del codice. Problema Scrivere una funzione che restituisca una lista di numeri primi ricavata con il metodo del crivello di Eratostene. E una funzione che, partendo da una lista vuota, aggiunga man mano numeri primi. Soluzione Si fa risalire ad Eratostene, un metodo furbo per individuare i numeri primi presenti nella sequenza dei primi numeri naturali: Si scrivono i primi enne numeri naturali, Si toglie lo zero e l'uno che non sono primi Si tiene il prossimo numero (in questo caso il 2) e si cancellano tutti i suoi multipli, Si

ripete l'istruzione precedente fino ad essere arrivati in fondo. Di seguito riporto i risultati delle prime iterazioni dell'algoritmo descritto sopra.

In un linguaggio di progetto il la funzione può diventare: def crivello(n):

popola una lista con i numeri da 2 a n per ogni elemento della lista,

se divide un elemento che lo segue cancella dalla lista quest'altro elemento

..restituisci la lista Per realizzare l'algoritmo ci sarà bisogno di due cicli annidati e di due indici, uno per scorrere tutti gli elementi della lista, l'altro per realizzare il confronto di questi elementi con gli altri: def crivello(n):

c=range(2, n+1) print "prima dei buchi:n", c i=0 while i<len(c):

j=i+1 while j<len(c):

print c[i], c[j] if c[j] % c[i] == 0: del c[j] j+=1

i+=1

print "ndopo i buchi:n", c return c

Come il solito, qualche istruzione print ben piazzata può contribuire a capire il funzionamento del codice, una volta esaurita la sua funzione, può essere commentata o cancellata. Il crivello funziona per "sottrazione", metto tutti i numeri da 2 a enne e poi elimino i multipli, posso anche operare per "addizione", parto da una lista vuota e aggiungo i numeri primi. Questa funzione va bene se voglio avere una lista con numeri primi inferiori ad un certo valore, ma se volessi generare la sequenza dei numeri primi dovrei procedere per addizione. In pseudocodice: def primi(massimo):

preparo un lista vuota per ogni numero da 2 al massimo:

se è divisibile per un numero compreso tra 2 e il numero stesso: non è primo

altrimenti: aggiungo questo numero alla lista dei numeri primi

restituisco la lista

Per tradurre questo algoritmo in Python dobbiamo predisporre alcune variabili e realizzare due cicli, uno annidato nell'altro: def primi0(massimo):

primi=[] num=1 while num<massimo:

num+=1 d=2 while d<num:

if (num % d)==0: break d+=1

if not d<num: primi.append(num) print num

return primi

Quando un algoritmo contiene dei cicli annidati è buona norma dubitare che sia efficiente. Le istruzioni contenute nel ciclo più interno verranno eseguite molte volte. Nella funzione precedente per trovare i numeri primi inferiori a 200, l'operazione che calcola il resto della divisione:

num % d, viene eseguita 4424 volte. C'è spazio per migliorare l'algoritmo... Intanto possiamo osservare che a parte il numero 2 tutti gli altri primi sono numeri dispari quindi, possiamo trattare 2 come caso particolare e poi cercare i primi solo tra i numeri dispari incrementando il valore di num di due invece che di uno. Chiamiamo i l'incremento. Ovviamente non occorrerà più verificare se un numero è divisibile per un numero pari, quindi anche il possibile divisore andrà incrementato di 2: def primi1(massimo):

primi=[2] i=2 print 2 num=1 while num<massimo:

num+=i d=3 while d<num:

if (num % d)==0: break d+=i

if not d<num: primi.append(num) print num

return primi

Quest'altro algoritmo, nelle stesse condizioni della prova precedente, esegue 2141 volte l'istruzione: num % d. Con questi semplici cambiamenti abbiamo più che dimezzato le istruzioni eseguite dall'algoritmo. Ma si può fare di più. Nessun numero è divisibile per un valore compreso tra la metà del numero stesso e il numero stesso. Ad esempio nell'intervallo]26; 53[non possono esserci divisori di 53. Quindi se non ho trovato un divisore tra 3 e enne/2 non lo troverò neppure tra enne/2 e enne-1. Quindi quando il divisore supera la metà del numero, posso sospendere la ricerca. Cambia la condizione che controlla il ciclo while e cambia di conseguenza il controllo per riconoscere se un numero è primo: def primi2(massimo):

primi=[2] i=2 print 2 num=1 while num<massimo:

num+=i d=3 while d<(num//2):

if (num % d)==0: break d+=i

if not d<(num//2): primi.append(num) print num

return primi

In questo modo, le operazioni più critiche (% e //) vengono eseguite 1285 volte: ancora un buon miglioramento! Ma possiamo ridurre ancora il numero dei controlli, infatti se osserviamo bene il comportamento dei numeri possiamo scoprire che se c'è un divisore maggiore della radice quadrata del numero allora deve essercene anche uno minore. Se 100 è divisibile per un numero maggiore di 10, supponiamo 25 allora deve essere divisibile anche per un numero inferiore a 10, infatti è divisibile per 100/25 che è 4. Viceversa se non abbiamo trovato divisori di enne nell'intervallo tra 3 e la radice quadrata di enne, non ce ne saranno neppure dopo. il nostro ciclo più interno può quindi fermarsi alla radice quadrata del numero invece che andare fino alla metà del numero, un altro bel risparmio di cicli. In Python, la radice quadrata non è una funzione interna, si trova nella libreria math quindi per poterla usare dobbiamo caricare la libreria con l'istruzione: import math e per eseguirla dobbiamo scrivere: math.sqrt(num). L'algoritmo diventa: def primi3(massimo):

import math primi=[2] i=2 print 2 num=1 while num<massimo:

num+=i d=3 radq=math.sqrt(num) while d<radq:

if (num % d)==0: break d+=i

if not d<=radq: primi.append(num) print num

return primi

Poiché anche il calcolo della radice quadrata di un numero è un'operazione pesante, invece di farla eseguire due volte, nella condizione del ciclo while e nell'istruzione if, ho preferito utilizzare una variabile. sommando le chiamate a math.sqrt e all'operatore che calcola il resto della divisione(%), ottengo 448 istruzioni eseguite. Circa un terzo della versione precedente e un decimo di quella iniziale! Cambiando un po' l'algoritmo si può migliorare ancora... Leggere una libreria matematica per eseguire un'operazione, non è il massimo dell'efficienza e la radice quadrata stessa è un'operazione piuttosto pesante. Python ci mette a disposizione una funzione che, dati un dividendo e un divisore, restituisce, come risultato, il quoziente e il resto della divisione.: >>> print divmod(27, 4) (6, 3) perché 27 equivale a 6*4+3. divmod() è una funzione molto efficiente e possiamo utilizzarla nel nostro algoritmo. Dall'osservazione che se divido enne per un numero inferiore alla sua radice quadrata ottengo un quoziente maggiore della sua radice quadrata, possiamo trasformare il ciclo più interno in modo che calcoli quoziente e resto della divisione tra num e d e termini in uno di questi due casi: se il resto è uguale a 0, in questo caso num non è primo, o se il divisore ha superato il quoziente, in questo caso num è primo. Per uscire da un ciclo Python mette a disposizione l'istruzione break. Quindi l'algoritmo diventa: def primi4(massimo):

primi=[2, 3] i=2 print 2 print 3 num=3 while num<massimo:

num+=i d=3 while True:

q, r = divmod(num, d) if r==0:

break

d+=i if d>q :

primi.append(num) print num break

return primi

Il comando while True: indica un ciclo che continua finché non viene eseguita un'istruzione break. Il numero di divisioni, chiamate alla funzione divmod() ora sono: 278, un altro notevole miglioramento. Con qualche altro trucco si può migliorare ulteriormente l'efficienza dell'algoritmo.

Riassumendo Per calcolare se un numero è divisibile per un altro si può usare l'operatore % oppure la funzione divmod()verificando che il resto sia zero. Si può terminare un ciclo usando l'istruzione break. Scrivere un programma che funziona è il primo passo, il programma deve anche essere "elegante " ed efficiente.

2.6. I numeri dell'orologio Come produrre le tabelline delle 4 operazioni con le classi di resto modulo enne. Prerequisiti operatori: %, <, e ==. funzioni con parametri, struttura di selezione: if ..else, struttura di iterazione: while e istruzione break, funzione divmod() liste.

Argomenti trattati classi resto, visualizzazione di una matrice,

Problema Scrivere delle funzioni che visualizzino le tabelline delle quattro operazioni con i numeri modulo enne.

Soluzione Quanti sono i numeri? Domanda alla quale, fin dalle elementari, si impara a rispondere: "infiniti"! Infatti, a qualunque numero si arrivi basta aggiungere "1" per ottenerne un altro. Questo ragionamento fila se pensiamo che aggiungendo "1" ad un numero si ottenga

sempre un numero nuovo non presente nelle sequenza di numeri già incontrati. Si possono pensare, invece, degli insiemi di numeri nei quali, ad un certo punto, aggiungendo "1" si ottenga di nuovo "0": . In questo modo i numeri si avvolgono su sé stessi e si ottiene un insieme limitato di numeri. Nel caso precedente l'insieme è costituito da 6 numeri: {0, 1, 2, 3,4, 5}. Poiché questi numeri possono essere visti come tutti i possibili resti della divisione di un numero intero qualunque per 6, sono anche detti "classi di resto modulo 6". Ora, si potrebbe pensare che un insieme di numeri così limitato possa risultare del tutto inutile, ma non è così. Questi insiemi finiti di numeri sono stati studiati dal grande matematico C.F. Gauss. È possibile definire alcune operazioni su questi insiemi di numeri. Le operazioni definite su questi numeri hanno delle proprietà molto interessanti tanto è vero che attualmente la crittografia e quindi tutto ciò che riguarda la sicurezza, nella nostra società, è basato su queste proprietà, oltre che sulle proprietà dei numeri primi e della scomposizione in fattori primi. Messa giù così, si potrebbe pensare che questa faccenda riguardi soltanto problemi molto più grandi di noi, invece noi usiamo tutti i giorni i numeri di questo tipo, precisamente quando guardiamo l'orologio o quando facciamo dei conti con le ore: 5 ore dopo le 9 della mattina sono le 2 del pomeriggio: 9+5=2. I numeri dell'orologio sono le classi di resto modulo 12 o modulo 24 a seconda di come abbiamo settato la nostra sveglia. Per ottenere il risultato di un'operazione in questi insiemi di numeri, possiamo eseguire l'operazione come avessimo a che fare con i normali numeri interi e poi dividere il risultato per il modulo (nel caso delle ore: 12 (o 24)) e tenere il resto: 912+512=resto_della_divisione(9+5, 12) Cioè: se alla "classe di resto modulo 12" 9 aggiungo la "classe di resto modulo 12" 5 ottengo il resto della divisione tra la somma 9 + 5 e 12. A rileggere il testo in "simil italiano" mi pare così complicato! Che grande invenzione i simboli matematici!!! Veniamo a noi, vogliamo scrivere un programma che costruisca e stampi le tabelline delle operazioni con i numeri classe resto modulo enne. Prima di tutto precisiamo come può essere rappresentata all'interno di Python una tabella quadrata come la seguente: 2 3 4 1 2 3 4 3 2 Possiamo usare le liste: una lista di 3 numeri rappresenta una riga e tre liste rappresentano l'intera matrice: matrice=[[2, 3, 4], [1, 2, 3], [4, 3, 2]] Siccome la rappresentazione con i numeri ben schierati è più leggibile di una lista di liste, prima di procedere ci costruiamo una funzione che stampi per bene una matrice con un numero qualunque di righe e di colonne. In pseudocodice: def printmat(matrice):

per ogni riga contenuta nella matrice:

per ogni elemento contenuto nella riga: stampa l'elemento senza andare a capo

va a capo

Dato che i numeri delle matrici che costruiremo saranno sempre compresi tra 0 e 99, cioè avranno al massimo 2 cifre, per il comando stampa conviene usare la stringa di formattazione "%2s" che produce una stringa che contiene il numero formata sempre da due caratteri, eventualmente aggiungendo uno spazio prima se il numero è minore di 10. Tradotto in Python: def printmat(mat):

""Stampa una matrice bidimensionale di numeri di una o due cifre allineando le colonne.""

for riga in mat:

for elemento in riga: print "%2s" % elemento,

print

Notare che la virgola che termina il primo comando print sta a significare: “stampa senza andare a capo”. Dopo aver salvato il file e averlo eseguito, proviamo il funzionamento: >>> m=[[2, 3, 4], [1, 2, 3], [4, 3, 2]] >>> printmat(m) 2 3 4 1 2 3 4 3 2 Ora che ci siamo forniti dello strumento per stampare una matrice di numeri concentriamoci sul problema: vogliamo scrivere una funzione che produca la tabellina (matrice) dei risultati delle operazioni con le classi modulo resto enne. Iniziamo dalla moltiplicazione. In pseudocodice: def matprod(mod):

predisponi una matrice vuota per mod volte:

predisponi una riga vuota per mod volte:

calcola il prossimo ele. della lista e agg. alla riga

aggiungi la riga alla matrice

restituisci la matrice

Proviamo a vedere un caso abbastanza semplice e poi a generalizzarlo:

x 0 1 2 3 0 0 0 0 0 1 0 1 1 3 2 0 2 0 2 3 0 3 2 1

Le formule per ottenerlo sono: x 0 1 2 3 0 0*0 mod 4 0*1 mod 4 0*2 mod 4 0*3 mod 4 1 1*0 mod 4 1*1 mod 4 1*2 mod 4 1*3 mod 4 2 2*0 mod 4 2*1 mod 4 2*2 mod 4 2*3 mod 4 3 3*0 mod 4 3*1 mod 4 3*2 mod 4 3*3 mod 4

Per ottenere il valore di una cella devo prendere l’intestazione della riga di quella cella (indice della riga), moltiplicarla per l’intestazione della colonna di quella cella (l’indice della colonna) e calcolare il resto della divisione per 4 se sto lavorando con le classi di resto modulo 4. Scegliamo un nome per gli indici delle righe: iy e un nome per gli indici delle colonne: ix (sarebbe andato altrettanto bene ir, ic o meglio indice_riga, indice_colonna) e raffiniamo l’algoritmo che produce la tabella della moltiplicazione: def matprod(mod):

predisponi una matrice vuota per iy che indica ogni riga:

predisponi una riga vuota per ix che indica ogni colonna:

aggiungi alla riga: ix*iy % mod

aggiungi la riga alla matrice

restituisci la matrice

Prova a tradurlo in Python prima di proseguire con la lettura... E confrontalo con: def matprod(mod):

“”“Produce la tabellina della molt. dei numeri Zmod. Metodo che utilizza l’iterazione for.”“” mat=[] for iy in range(mod):

riga=[] for ix in range(mod):

riga.append(ix*iy%mod)

mat.append(riga)

return mat

Aggiungendo in fondo al programma le righe seguenti: def main():

printmat(matprod(12)) print

if __name__==”__main__”: main() ed eseguendo il programma, si dovrebbe ottenere quanto voluto. Cambiando l’argomento passato alla funzione si ottengono le tabelline della moltiplicazione delle diverse classi di resto modulo enne. Scrivere un programma che funziona è il primo passo. Il secondo è sistemare il codice in modo che sia più efficiente o più leggibile. Passiamo alla seconda fase. Il secondo ciclo for, quello più interno, produce una lista. Python ha un meccanismo di costruzione di liste (detto list comprehension) che permette di riassumere tutte le informazioni contenute nelle 4 righe:

```
riga=[] for ix in range(mod):
    riga.append(ix*iy%mod)
mat.append(riga)
```

in un’unica riga: mat.append([ix*iy%mod for ix in range(mod)])

Chi programma Python ritiene che questo modo di programmare sia più “Pitonico”, più semplice e comprensibile. Ovviamente bisogna farci un po’ l’abitudine, come in tutte le cose. La funzione allora diventa: def matprod(mod):

```
“”“Produce la tabellina della molt. dei numeri Zmod. Metodo che usa l’iterazione
for e la costr. di liste.”“” mat=[] for iy in range(mod):
    mat.append([ix*iy%mod for ix in range(mod)])
return mat
```

A questo punto i più scaltri avranno riconosciuto che il ciclo for, e l’istruzione che la precede, serve per costruire una lista, possiamo applicare ancora il meccanismo della list comprehension ed ottenere una funzione costituita da una sola riga: def matprod(mod):

```
“”“Produce la tabellina della molt. dei numeri Zmod. Metodo che usa la
costruzione di liste.”“” return [[ix*iy%mod for ix in range(mod)] for iy in
    range(mod)]
```

Un codice più compatto, spesso è più semplice da verificare e da correggere, ma son si deve mai sacrificare la chiarezza e la leggibilità per la sintesi: nella scala dei valori, la sintesi è importante, ma la leggibilità lo è di più!

Riassumendo Posso memorizzare una matrice rettangolare sotto forma di lista di liste. Posso scrivere una funzione che stampi in modo ordinato una matrice. Usando l’operatore “%” posso calcolare il resto di una divisione. In Python la costruzione di liste (list comprehension) permette di sostituire un ciclo for sintetizzando 4 righe in una.

2.7. Scomposizione in fattori primi Come scomporre un numero in fattori primi. Prerequisiti operatori: >, e ==. funzioni con parametri, funzione divmod() struttura di selezione: if ..else, struttura di iterazione: while e istruzione break, liste e tuple.

Argomenti trattati scomposizione di un numero in fattori primi, programmazione guidata dai test,

Problema Scrivere una funzione che produca la scomposizione in fattori primi di un numero.

Soluzione Il teorema fondamentale dell'aritmetica dice che la scomposizione in fattori primi di un numero è unica. Ciò vuol dire che posso seguire diverse strade, ma il risultato dipende solo dal numero e non dal percorso seguito:

```
 72 = 23 32       72 = 23 32       72 = 23 32
 / \     / \      / \
2  36           4  18           8   9
   / \         / \ / \         / \ / \
  6   6       2  2 2  9       2  4 3  3
 / \ / \             / \       / \
2  3 2  3           3   3     2   2
```

Il metodo che preferisco, per scomporre in fattori un numero, è quello dell'albero binario:

se il numero è primo, è già scomposto in fattori primi, altrimenti da questo numero derivo due rami sotto cui scrivo due numeri il cui prodotto dia il numero di partenza, ripeto i due passi precedenti per ogni nuovo numero.

Questo metodo, ricorsivo, permette ad ogni passo di scegliere i divisori più comodi e quindi in molti casi rende più semplice il calcolo rispetto al metodo che solitamente si trova nei libri delle medie:

```
3600 | 2              3600 = 243252
1800 | 2             /    \
 900 | 2          100      36
 450 | 2         /  \     /  \
 225 | 3        10   10   4   9
  75 | 3       / \   / \ / \ / \
  25 | 5       2 5   2 5 2 2 3 3
   5 | 5
   1 |
3600 = 243252
```

A me piace anche perché non devo riscrivere il numero di partenza quando ho terminato la scomposizione (la solita pigrizia). Il metodo ad albero binario diventa del tutto uguale al metodo proposto sui libri, se ogni volta scelgo il divisore più piccolo possibile:

```
3600 = 243252
 / \
2   1800
     / \
    2   900
        / \
       2   450
           / \
          2   225
              / \
             3  75
                / \
               3  25
                  / \
                 5   5
```

Il metodo ad albero prevede che siamo noi a scegliere tra tutti i possibili divisori di un numero quale utilizzare, il teorema fondamentale dell'aritmetica ci assicura che qualunque divisore va bene. Se la scelta la facciamo con un po' di buon senso possiamo renderci più semplice il calcolo. Ora, se vogliamo realizzare una funzione automatica che scomponga in fattori un numero, dobbiamo tenere presente che, per le macchine (purtroppo spesso anche per i politici e gli amministratori in genere), il buon senso risulta molto difficile, mentre risulta facile il calcolo. Quindi l'algoritmo può diventare qualcosa di simile a questo: Dividi il num. per il più piccolo divisore (maggiore di 1!) dividi il quoziente ottenuto per il più piccolo divisore continua così finché non hai ottenuto come quoziente 1 Nel caso del numero 120 otteniamo:

```
120 | 2
 60 | 2
 30 | 2
 15 | 3
  5 | 5
  1 |
```

Abbiamo visto che non è il metodo più pratico quando dobbiamo fare a mano la scomposizione, ma è molto meccanico e quindi è più facilmente trasformabile in un algoritmo. Otteniamo quindi . Prima di procedere dobbiamo pensare ad una struttura di dati che permetta di rappresentare tutti i possibili risultati delle scomposizioni in fattori: ma anchee anchee ... Perché scrivere potenze separate da un puntino è comodo per un essere umano, ma non lo è molto per un computer. Ricordando che ,..., ogni fattore può essere visto come una potenza, possiamo osservare che la scomposizione in fattori primi è formata da una sequenza di potenze in numero variabile. In Python la struttura di dati che viene utilizzata per contenere sequenze di lunghezza variabile è la lista, quindi la funzione che scomporrà in fattori un numero dovrà restituire come risultato una lista di potenze: [potenza1, potenza2, potenza3, ...]. Una potenza può essere vista come una coppia di numeri: la base e l'esponente. Una coppia in Python può essere rappresentata da una tupla: 23 = (2, 3), 51 = (5, 1), ... La scomposizione in fattori di un numero può essere vista come una sequenza di potenze ed essere rappresentata in Python come una lista di tuple: la rappresentazione interna di può essere la lista di coppie: [(2, 3), (3, 1), (5, 1)] che va interpretata come due elevato alla terza per tre elevato alla prima per cinque elevato alla prima. Ora abbiamo tutti gli elementi per affrontare il problema: incominciamo a scrive il codice! Il metodo seguito di solito è quello di:

1. scrivere una prima funzione
2. provarla in vari casi
3. correggerla
4. tornare al punto 2 finché non risulta sufficientemente corretta.

Questo metodo funziona, ma dal punto di vista informatico è molto meglio, prima scrivere i test del nostro programma e poi il codice che risolve il problema: prima definiamo gli "esercizi" e i risultati che ci aspettiamo di ottenere e poi scriviamo la funzione che dovrà risolvere gli "esercizi". Questo modo di procedere, che può sembrare strano, è così importante da meritare un nome: "programmazione guidata dai test", "test driven". Bene, mano all'editor e iniziamo a scrivere il nostro programma: def fattorizza(numero):

"""Scomposizione in fattori primi di numero.""" risultato = [] return risultato

def prova(funzione, risultato_atteso):

“”“Esegue la funzione e confronta il risultato con risultato_atteso.”“”

risultato=eval(funzione) if risultato==risultato_atteso:

print “%s -> %s corretto” % (funzione, risultato)

else:

print “%s -> %s, risultato atteso: %s” % (funzione, risultato, risultato_atteso)

def test(): prova(“fattorizza(7)”, [(7, 1)]) prova(“fattorizza(8)”, [(2, 3)]) prova(“fattorizza(23)”, [(23, 1)]) prova(“fattorizza(28)”, [(2, 2), (7, 1)]) prova(“fattorizza(30)”, [(2, 1), (3, 1), (5, 1)]) prova(“fattorizza(120)”, [(2, 3), (3, 1), (5, 1)]) prova(“fattorizza(%s)” % (2*2*3*3*3*3*7*11*11),

[(2, 2), (3, 4), (7, 1), (11, 2)])

if __name__==”__main__”: test() Qualche commento alle 3 funzioni. La prima, fattorizza(), è quella che dobbiamo costruire. È definita in modo decisamente stupido: qualunque sia l’argomento passato alla funzione, e messo nel parametro numero, darà come risultato una lista vuota. Per ora va bene così perché ci concentriamo sulle funzioni che servono per il test. La funzione prova() è la più complessa: Ha 2 parametri, nel primo viene messa la funzione da eseguire e nel secondo il risultato atteso. Esegue la funzione e memorizza il risultato. Confronta il risultato ottenuto con quello atteso e stampa un messaggio adeguato. L’ultima, test() è costituita da una sequenza di chiamate alla funzione prova() passandole via via diverse funzioni da verificare con i risultati attesi. Quando eseguiamo il programma l’ultima linea chiama la funzione test() e otteniamo il seguente output: fattorizza(7) -> [], risultato atteso: [(7, 1)] fattorizza(8) -> [], risultato atteso: [(2, 3)] fattorizza(23) -> [], risultato atteso: [(23, 1)] fattorizza(28) -> [], risultato atteso: [(2, 2), (7, 1)] fattorizza(30) -> [], risultato atteso: [(2, 1), (3, 1), (5, 1)] fattorizza(120) -> [], risultato atteso: [(2, 3), (3, 1), (5, 1)] fattorizza(274428) -> [], risultato atteso: [(2, 2), (3, 4), (7, 1), (11, 2)] A questo punto possiamo essere moderatamente soddisfatti: il programma non fa quello che vogliamo, ma funziona correttamente. Un piccolo cambiamento alla funzione fattorizza() le permette di trattare correttamente i numeri primi:

def fattorizza(numero): “”“Scomposizione in fattori primi di numero.”“” risultato = [(numero, 1)] return risultato

L’output è: fattorizza(7) -> [(7, 1)] corretto fattorizza(8) -> [(8, 1)], risultato atteso: [(2, 3)] fattorizza(23) -> [(23, 1)] corretto fattorizza(28) -> [(28, 1)], risultato atteso: [(2, 2), (7, 1)] fattorizza(30) -> [(30, 1)], risultato atteso: [(2, 1), (3, 1), (5, 1)] fattorizza(120) -> [(120, 1)], risultato atteso: [(2, 3), (3, 1), (5, 1)] fattorizza(274428) -> [(274428, 1)], risultato atteso: [(2, 2), (3, 4), (7, 1), (11, 2)] I numeri 7 e 23 vengono scomposti correttamente e questo ci conforta sul buon funzionamento delle funzioni di test che, oltre a riconoscere i casi sbagliati, tutti nella prima versione di fattorizza(), riconoscono anche quelli corretti. A questo punto conviene riprendere gli esempi di scomposizione in fattori risolti con carta e penna. È chiaro che ci sono delle operazioni che vengono ripetute fin quando il numero da scomporre è maggiore di 1. In informatica ciò si traduce con un ciclo while: fin quando numero>1:

esegui qualcosa

Ma prima di iniziare il ciclo devo porre il divisore uguale al più basso valore, cioè porre il divisore uguale a 2. divisore=2 fin quando numero>1:

esegui qualcosa

All'interno del ciclo devo calcolare quoziente e resto della divisione di numero per divisore e controllare il resto. Se il resto è uguale a zero, aggiungo una coppia e aggiorno il numero, altrimenti incremento il divisore: divisore=2 fin quando numero>1:

calcolo quoziente e resto di numero / divisore se resto è uguale a 0:

aggiungo una coppia al risultato aggiorno il numero

altrimenti incremento il divisore

Le idee, a questo punto sono abbastanza chiare per poter essere tradotte in Python: def fattorizza(numero):

"""Scomposizione in fattori primi di numero.""" risultato = [] divisore=2 while numero>1:

quoziente, resto = divmod(numero, divisore) if resto==0: # numero è multiplo di divisore

risultato.append((divisore, 1)) # aggiungo una coppia a risultato

numero=quoziente # aggiorno numero

else: divisore+=1 # incremento divisore

return risultato

L'esecuzione del programma produce: fattorizza(7) -> [(7, 1)] corretto fattorizza(8) -> [(2, 1), (2, 1), (2, 1)], risultato atteso: [(2, 3)] fattorizza(23) -> [(23, 1)] corretto fattorizza(28) -> [(2, 1), (2, 1), (7, 1)], risultato atteso: [(2, 2), (7, 1)] fattorizza(30) -> [(2, 1), (3, 1), (5, 1)] corretto fattorizza(120) -> [(2, 1), (2, 1), (2, 1), (3, 1), (5, 1)], risultato atteso: [(2, 3), (3, 1), (5, 1)] fattorizza(274428) -> [(2, 1), (2, 1), (3, 1), (3, 1), (3, 1), (3, 1), (7, 1), (11, 1), (11, 1)], risultato atteso: [(2, 2), (3, 4), (7, 1), (11, 2)] Ora anche 30 è fattorizzato correttamente, non solo, anche le altre soluzioni sarebbero giuste se noi avessimo deciso di accettare al posto di . Rimbocchiamoci le maniche e cerchiamo di aggiustare le cose. Nel precedente algoritmo effettivamente non c'è traccia di esponenti. Prima di iniziare il ciclo oltre a porre divisore uguale a 2 dobbiamo porre l'esponente uguale a 0 e aumentare l'esponente ogni volta che divisore divide numero. Quando divisore divide numero, invece di aggiungere una coppia, incrementiamo l'esponente. La coppia viene invece aggiunta quando divisore non divide numero solo se l'esponente è maggiore di zero: def fattorizza(numero):

"""Scomposizione in fattori primi di numero.""" risultato = [] divisore=2 esponente=0 while numero>1:

quoziente, resto = divmod(numero, divisore) if resto==0: # numero è multiplo di divisore

esponente+=1 # incremento esponente numero=quoziente # aggiorno numero

else:

if esponente>0: # se esponente è cambiato

risultato.append((divisore, esponente)) # agg. una # coppia a ris.

divisore+=1 # incremento divisore

return risultato

Operate le modifiche basta eseguire il programma: fattorizza(7) -> [], risultato atteso: [(7, 1)] fattorizza(8) -> [], risultato atteso: [(2, 3)] fattorizza(23) -> [], risultato atteso: [(23, 1)] fattorizza(28) -> [(2, 2), (3, 2), (4, 2), (5, 2), (6, 2)],

risultato atteso: [(2, 2), (7, 1)]

fattorizza(30) -> [(2, 1), (3, 2), (4, 2)], risultato atteso: [(2, 1), (3, 1), (5, 1)] fattorizza(120) -> [(2, 3), (3, 4), (4, 4)], risultato atteso: [(2, 3), (3, 1), (5, 1)] fattorizza(274428) -> [(2, 2), (3, 6), (4, 6), (5, 6), (6, 6), (7, 7), (8, 7), (9, 7), (10, 7)], risultato atteso:

[(2, 2), (3, 4), (7, 1), (11, 2)]

Disastro! Anche quello che prima funzionava ora non va più! Cosa è successo? Possiamo fare una prima osservazione: nei numeri primi il risultato è una lista vuota, perché finito il ciclo ci è rimasta in mano una coppia che dobbiamo aggiungere alla lista. Quindi, prima di restituire il risultato dobbiamo aggiungere l'ultima coppia: def fattorizza(numero):

"""Scomposizione in fattori primi di numero.""" risultato = [] divisore=2 esponente=0 while numero>1:

quoziente, resto = divmod(numero, divisore)

 # print numero, divisore, quoziente, resto

if resto==0: # numero è multiplo di divisore esponente+=1 # incremento esponente numero=quoziente # aggiorno numero

else:

if esponente>0: # se esponente è cambiato

risultato.append((divisore, esponente)) # agg. una # coppia a ris.

divisore+=1 # incremento divisore

risultato.append((divisore, esponente)) # agg. una coppia # a risultato

return risultato

Ora il programma produce: fattorizza(7) -> [(7, 1)] corretto fattorizza(8) -> [(2, 3)] corretto fattorizza(23) -> [(23, 1)] corretto fattorizza(28) -> [(2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (7, 3)], risultato atteso: [(2, 2), (7, 1)] fattorizza(30) -> [(2, 1), (3, 2), (4, 2), (5, 3)], risultato atteso: [(2, 1), (3, 1), (5, 1)] fattorizza(120) -> [(2, 3), (3, 4), (4, 4), (5, 5)], risultato atteso: [(2, 3), (3, 1), (5, 1)] fattorizza(274428) -> [(2, 2), (3, 6), (4, 6), (5, 6), (6, 6), (7, 7), (8, 7), (9, 7), (10, 7), (11, 9)], risultato atteso: [(2, 2), (3, 4), (7, 1), (11, 2)] Qualcosa è migliorato: i numeri con un solo fattore primo, non importa con quale esponente, sono fattorizzati in modo corretto, gli altri sono strampalati. Il problema deriva dal fatto che, all'interno del ciclo, quando viene aggiunta una coppia al risultato, l'esponente deve essere rimesso a zero. def fattorizza(numero):

“”“Scomposizione in fattori primi di numero.”“” risultato = [] divisore=2 esponente=0 while numero>1:

quoziente, resto = divmod(numero, divisore) if resto==0: # numero è multiplo di divisore

esponente+=1 # incremento esponente numero=quoziente # aggiorno numero

else:

if esponente>0: # se esponente è cambiato

risultato.append((divisore, esponente)) # agg. una # coppia a ris.

esponente=0 # rimette a zero esponente

divisore+=1 # incremento divisore

risultato.append((divisore, esponente)) # agg. una coppia # a risultato

return risultato

Il programma ora produce: fattorizza(8) -> [(2, 3)] corretto fattorizza(23) -> [(23, 1)] corretto fattorizza(28) -> [(2, 2), (7, 1)] corretto fattorizza(30) -> [(2, 1), (3, 1), (5, 1)] corretto fattorizza(120) -> [(2, 3), (3, 1), (5, 1)] corretto fattorizza(274428) -> [(2, 2), (3, 4), (7, 1), (11, 2)] corretto Se la nostra base di test è abbastanza ampia possiamo ritenerci soddisfatti. Non solo, se qualche utente della nostra funzione, o noi stessi ci accorgiamo di un particolare caso che produce un errore, basta che lo aggiungiamo ai casi da verificare e poi modificare la fattorizza() in modo che funzioni correttamente in tutta la nostra base di prove. Questo meccanismo permette di evitare che la correzione di un errore produca un errore nei casi che prima funzionavano correttamente.

Riassumendo Un buon modo di scrivere un programma è quello di farsi guidare dai test. In particolare nei programmi complessi, perdere del tempo nella costruzione di una adeguata batteria di test fa guadagnare, complessivamente, molto tempo. Prima di mettersi a scrive del codice si deve arrivare a dare una buona descrizione a parole dell’algoritmo che si vuole realizzare. Se l’algoritmo è pensato bene, non è difficile correggere eventuali errori magari scovandoli con qualche istruzione print posizionata appropriatamente.

CAPITOLO 3

La geometria interattiva

3.1 Introduzione

Cos'è la geometria interattiva, i primi oggetti.

La geometria interattiva permette di creare gli oggetti della geometria euclidea:

- punti;
- rette;
- circonferenze.

I punti possono essere:

- liberi;
- vincolati a una linea;
- intersezioni di due linee.

Le rette possono essere anche:

- semirette;
- segmenti.

I punti base possono essere trascinati con il mouse quindi, se ho realizzato una costruzione geometrica a partire da da alcuni punti, quando muovo questi punti tutta la costruzione si muove.

La Geometria interattiva mette in evidenza quali sono le caratteristiche invarianti e quali quelle variabili di una certa costruzione.

Esistono molti programmi che permettono di operare con la geometria interattiva, In questo testo propongo l'uso del linguaggio Python con la libreria `pyig`.

È possibile seguire il percorso proposto anche con un programma *punta e clicca* invece che con un linguaggio.

Per l'installazione di Python e della pibreria Pygraph, che contiene anche Pyig, vedi l'introduzione alla geometria della tartaruga nel volume precedente.

Riassumendo

- La geometria interattiva permette di creare e di muovere gli oggetti della geometria euclidea.
- Ci sono molti programmi che permettono di giocare con la geometria interattiva, noi utilizzeremo il linguaggio Python con la libreria `pyig`.
- Gli oggetti di base sono:
  - punti: * liberi, * vincolati, * intersezioni;
  - rette: * rette, * semirette, * segmenti;
  - circonferenze.

Prova tu

1. Installa Python.
2. Installa la libreria pygraph.

3.2 Elementi fondamentali

Come creare un piano vuoto, dei punti, delle rette e altri oggetti geometrici.

La geometria si basa su alcuni *elementi fondamentali* che non si definiscono: *piano*, *punto*, *retta*. A partire da questi oggetti se ne possono definire altri.

I *postulati* definiscono le relazioni tra gli elementi fondamentali.

Le *figure geometriche* sono insiemi di oggetti geometrici.

La *geometria* studia le caratteristiche che possono cambiare o che rimangono invarianti quando operiamo alcune trasformazioni. Ad esempio da un punto esterno ad una circonferenza posso tracciare due tangenti alla circonferenza stessa, se muovo il punto, la lunghezza delle tangenti varia, ma le due tangenti rimarranno sempre uguali tra di loro.

La]geometria interattiva] permette di visualizzare facilmente elementi varianti e invarianti di una certa costruzione geometrica.

3.2.1 Strumenti

In questo capitolo utilizzeremo i seguenti strumenti di Pyig:

- `Point(x, y)` crea un punto con date coordinate.
- `Line(p0, p1)` crea una retta passante per p0 e p1.
- `Ray(p0, p1)` crea una semiretta con origine in p0 passante per p1.
- `Segment(p0, p1)` crea un segmento di estremi p0 e p1.
- `Circle(centro, punto)` crea una circonferenza dati il centro e un suo punto.

3.2.2 Problema

Crea un piano e inserisci: un punto, una retta un segmento, una semiretta, un angolo, una circonferenza, un testo. Modifica poi le figure trascinando i punti base con il mouse.

Soluzione guidata

1. Crea un nuovo programma e salvarlo con il nome: `gi01Elementi.py`. Per creare un nuovo programma:
   (a) avvia IDLE (in Windows: menu-programmi-Python-IDLE);
   (b) crea un nuovo editor: menu-file-new window;
   (c) salvalo nella tua cartella con il nome desiderato: menu-file-save;

   #. esegui il programma in modo da controllare che non ci siano errori: tasto <F5>;
2. Incominciamo a scrivere il nostro programma.
   (a) Scrivi un'intestazione fatta da commenti che contenga le informazioni:
   - data,
   - nome,
   - titolo del programma;

   (a) esegui il programma in modo da controllare che non ci siano errori (<F5>).
3. Il programma vero e proprio è fatto da tre parti:
   (a) la lettura delle librerie;
   (b) il programma principale;
   (c) l'attivazione della finestra grafica.

   A questo punto il programma assomiglierà a:

```
# 7/9/14
# Daniele Zambelli
# Elementi di base della geometria

# lettura delle librerie

# programma principale

# attivazione della finestra grafica
```

4. Fin qui abbiamo scritto solo commenti, ora incominciamo a scrivere comandi:
   (a) Leggo la libreria Pyig e le do un nome più breve, "ig":

   import pyig

(a) Il programma principale consiste, per ora, in una sola istruzione, creo un "InteractivePlane" della libreria "ig" e associo questo oggetto all'identificatore (=alla parola) "ip":

ip = pyig.InteractivePlane()

(a) Rendo attiva la finestra grafica:

ip.mainloop()

5. Aggiungiamo le istruzioni sotto ai commenti:

```
# 7/9/14
# Daniele Zambelli
# Elementi di base della geometria

# lettura delle librerie
import pyig

# programma principale
ip = pyig.InteractivePlane()

# attivazione della finestra grafica
ip.mainloop()
```

6. Prova il programma premendo il tasto: `<Ctrl-c>` o cliccando su `menu-Run-Run module`. Deve apparire una finestra grafica con un riferimento cartesiano e una griglia di punti. La finestra grafica è attiva, risponde al mouse e si può chiudere. Se non avviene questo, probabilmente è apparso un messaggio di errore in rosso nella shell di `IDLE`, leggi il messaggio, correggi l'errore e ritenta.

Ora incominciamo ad aggiungere al programma principale le istruzioni per risolvere il problema. Incominciamo creando un punto. Aggiungiamo al programma principale il comando della libreria `pyig` che crea un punto associando l'oggetto appena creato all'identificatore "p_0":

```
p_0 = pyig.Point(3, 4)
```

È possibile trascinare con il mouse il punto nel suo piano: il punto non cambia cambiando la sua posizione.

In geometria un punto non dovrebbe avere altre caratteristiche oltre la propria posizione, ma a noi fa comodo poter dare ai punti anche altre caratteristiche come: uno spessore, un colore, un'etichetta:

```
p_1 = pyig.Point(-2, 6, color='red', width=6, name='A')
```

In generale a tutti gli oggetti di pyig che possono essere visualizzati si possono assegnare le seguenti caratteristiche:

- colore: color=<una stringa di colore>;
- spessore: width=<un numero>;
- etichetta: name=<una stringa>;
- visibilità: visible=<True> o <False>.

La sintassi del costruttore dell'oggetto `Point` è:

```
Point(<x>, <y>
      [, visible=True][, color='green'][, width=3][, name=''])
```

I primi due parametri, `x` e `y`, sono obbligatori, quelli messi tra parentesi quadre sono opzionali e, se non specificati, hanno il valore riportato sopra.

Passiamo alla seconda richiesta del problema: disegnare una retta.Per poter tracciare una retta abbiamo bisogno di due punti infatti due punti individuano univocamente una retta (per due punti passa una e una sola retta). Possiamo utilizzare i due punti già disegnati e creare la retta passante per `p_0` e `p_1`:

```
r_0 = pyig.Line(p_0, p_1)
```

E se vogliamo creare un'altra retta cambiando lo spessore, il colore e dandole un nome? Possiamo provare con la stessa tecnica usata per `Point`. Ma per disegnare un'altra retta ci servono altri due punti, dato che ci servono solo per creare la retta possiamo costruirli all'interno del costruttore della retta:

```
r_1 = pyig.Line(pyig.Point(-4, -3, color='blue', width=6, name='B'),
                pyig.Point(2, -6, color='green', width=6, name='C'),
                color='pink', width=4)
```

La sintassi del costruttore dell'oggetto `Line` è:

```
Line(<punto_0>, <punto_1>
      [, visible=True][, color='blue'][, width=3][, name=''])
```

È possibile trascinare con il mouse i punti base della retta, ma la retta continuerà a passare per quei due punti.

Le sintassi dei costruttori degli altri oggetti richiesti dal problema sono:

```
Ray(<punto_0>, <punto_1>
    [, visible=True][, color='blue'][, width=3][, name=''])

Segment(<punto_0>, <punto_1>
        [, visible=True][, color='blue'][, width=3][, name=''])

Circle(<centro>, <uunto>
       [, visible=True][, color='blue'][, width=3][, name=''])
```

Ora crea tu una semiretta, un segmento e una circonferenza.

Riassumendo

- Per lavorare con la geometria interattiva dobbiamo far caricare a Python la relativa *libreria* ad esempio con il comando:

```
    import pyig
```

- Un programma è composto (per ora) dalle seguenti parti:

```
<intestazione>
<lettura delle librerie>
<programma principale>
<attivazione della finestra grafica>
```

- La sintassi dei costruttori degli oggetti base della geometria è:

```
Point(<x>, <y>
      [, visible=True][, color='green'][, width=3][, name=''])

Line(<punto_0>, <punto_1>
     [, visible=True][, color='blue'][, width=3][, name=''])

Ray(<punto_0>, <punto_1>
    [, visible=True][, color='blue'][, width=3][, name=''])

Segment(<punto_0>, <punto_1>
        [, visible=True][, color='blue'][, width=3][, name=''])

Circle(<centro>, <punto>
       [, visible=True][, color='blue'][, width=3][, name=''])
```

Prova tu

1. Crea un nuovo programma che disegni un segmento di colore viola, con due estremi rosa, grandi a piacere.
2. Crea un programma che disegni un rettangolo. Muovendo i punti base
3. Crea un programma che disegni un triangolo. Muovendo i punti base continua a rimanere un triangolo? continua a rimanere un rettangolo?
4. Crea un programma che disegni un quadrilatero delimitato da semirette.
5. Crea un programma che disegni tre punti `A`, `B` e `C`, disegna poi le tre circonferenze:
   - di centro `A` e passante per `B`;
   - di centro `B` e passante per `C`;
   - di centro `C` e passante per `A`;
6. Disegna due circonferenze concentriche. Muovendo i punti base, si mantiene la proprietà "essere concentriche"?

3.3 Intersezioni

Come usare intersezioni tra oggetti.

Oltre a quelli visti nel capitolo precedente, per poter realizzare costruzioni geometriche abbiamo bisogno di poter creare l'intersezione tra due rette, tra una retta e una circondìferenza o tra due circonferenze.

3.3.1 Strumenti

In questo capitolo utilizzeremo i seguenti strumenti di Pyig:

- `Point(x, y)` crea un punto con date coordinate.
- `Line(p0, p1)` crea una retta passante per p0 e p1.
- `Segment(p0, p1)` crea un segmento di estremi p0 e p1.
- `Intersection(oggetto_0, oggetto_1, [which])` crea un punto di intersezione tra due oggetti.

3.3.2 Problema

Crea un piano e inserisci:

- due rette nel terzo e quarto quadrante e il segmento che congiunge la loro intersezione con l'origine;
- una retta e una circonferenza nel secondo quadrante e e i segmenti che congiungono le loro intersezioni con l'origine;
- due circonferenze nel primo quadrante e e i segmenti che congiungono le loro intersezione con l'origine;

Soluzione guidata

1. Crea un nuovo programma e salvarlo con il nome: `gi02Intersezioni.py`. Per creare un nuovo programma: vedi la soluzione guidata del capitolo precedente.
2. Scrivi un'intestazione adeguata.
3. Scrivi lo scheletro del programma:
   (a) la lettura delle librerie,
   (b) il programma principale,
   (c) l'attivazione della finestra grafica;
   e verifica che tutto funzioni.
4. Ora scriviamo dei commenti che indicano come intendiamo risolvere il problema:

```
# Creo le due rette
# Creo un punto nell'origine degli assi
# Creo l'intersezione tra le due rette
# Creo il segmento che congiunge l'origine all'intersezione
```

5. A questo punto il programma dovrebbe apparire circa così:

```
# 9/9/14
# Daniele Zambelli
# Intersezioni

# lettura delle librerie
import pyig as ig

# programma principale
ip = ig.InteractivePlane()

# Creo le due rette
# Creo un punto nell'origine degli assi
# Creo l'intersezione tra le due rette
# Creo il segmento che congiunge l'origine all'intersezione


# attivazione della finestra grafica
ip.mainloop()
```

6. Ora iniziamo a popolare di istruzioni il programma principale creando le due rette:

```
r_0 = ig.Line(ig.Point(-6, -4, width=6), ig.Point(2, -6, width=6))
r_1 = ig.Line(ig.Point(-11, -9, width=6), ig.Point(-3, -8, width=6))
```

   Eseguiamo il programma controllando che rispetti le specifiche.

7. Ora dobbiamo creare un segmento con un estremo nell'origine, quindi dobbiamo creare un punto nell'origine:

```
origine = ig.Point(0, 0, visible=False)
```

   e siccome vogliamo che nessuno possa muoverlo, lo facciamo invisibile.

8. L'altro estremo è l'intersezione delle due rette:

```
i_0 = ig.Intersection(r_0, r_1, color='red')
```

9. Infine creiamo il segmento:

```
s_0 = ig.Segment(origine, i_0, color='#505010')
```

10. A questo punto il programma dovrebbe apparire circa così:

```
# 9/9/14
# Daniele Zambelli
# Intersezioni

# lettura delle librerie
import pyig as ig

# programma principale
ip = ig.InteractivePlane()

# Creo le due rette
```

```
    r_0 = ig.Line(ig.Point(-6, -4, width=6), ig.Point(2, -6, width=6))
    r_1 = ig.Line(ig.Point(-11, -9, width=6), ig.Point(-3, -8, width=6))
    # Creo un punto nell'origine degli assi
    origine = ig.Point(0, 0, visible=False)
    # Creo l'intersezione tra le due rette
    i_0 = ig.Intersection(r_0, r_1, color='red')
    # Creo il segmento che congiunge l'origine all'intersezione
    s_0 = ig.Segment(origine, i_0, color='#505010')

    # attivazione della finestra grafica
    ip.mainloop()
```

11. Proviamo il programma e controlliamo che rispetti le specifiche richieste dal problema. Cosa succede quando muovo i punti base di una retta?

12. Se tutto funziona regolarmente possiamo passare alla seconda parte del problema:

```
    # Creo una retta
    # Creo una circonferenza
    # Creo le intersezioni tra la retta e la circonferenza
    # Creo i segmenti
```

13. Per quanto riguarda i primi due punti non dovrebbero esserci problemi, per il terzo invece presenta una novità rispetto a quanto visto per l'intersezione di due rette, infatti una retta interseca una circonferenza in due punti (e non sempre) e noi dobbiamo indicare a Python quale delle due intersezioni vogliamo:

```
    i_1 = ig.Intersection(r_2, c_0, -1, color='red')
    i_2 = ig.Intersection(r_2, c_0, +1, color='red')
```

14. Dopo aver controllato che fin qui il programma funzioni, disegniamo i due segmenti. la seconda parte dovrebbe assomigliare a questa:

```
    # Creo una retta
    r_2 = ig.Line(ig.Point(-11, 9, width=6), ig.Point(-6, 1, width=6))
    # Creo una circonferenza
    c_0 = ig.Circle(ig.Point(-6, 7), ig.Point(-5, 2))
    # Creo le intersezioni tra la retta e la circonferenza
    i_1 = ig.Intersection(r_2, c_0, -1, color='red')
    i_2 = ig.Intersection(r_2, c_0, +1, color='red')
    # Creo i segmenti
    s_1 = ig.Segment(origine, i_1, color='#10a010')
    s_2 = ig.Segment(origine, i_2, color='#10a080')
```

15. Proviamo il programma e controlliamo che rispetti le specifiche richieste dal problema. Cosa succede quando muovo i punti base della retta?

16. Completiamo il programma per risolvere anche la terza parte del problema.

Riassumendo

- pyig mette a disposizione un oggetto intersezione. L'intersezione tra rette non ha bisogno di ulteriori informazioni, quella tra una retta e una circonferenza o tra due circonferenze

richiede un ulteriore argomento: con `-1` si indica un'intersezione, con `+1` si indica l'altra. La sintassi del costruttore di un'intersezione è:

```
Intersection(oggetto_0, oggetto_1 [, which]
             [, visible=True] [, color='blue'] [, width=3] [, name=''])
```

Prova tu

1. Disegna una circonferenza `c_0` con il centro nell'origine, una retta `r_0` e un'altra circonferenza `c_1`. Disegna in modo evidente le intersezioni tra la retta `r_0` e la circonferenza `c_0` e tra la circonferenza `c_1` e la circonferenza `c_0`.
2. Disegna una circonferenza e una retta. Poi disegna un'intersezione tra la retta e la circonferenza e assegna a questa intersezione il nome: "Ciao". Poi disegna una circonferenza che ha centro nell'intersezione e passa per il punto (3; 1).
3. Disegna le intersezioni tra due circonferenze che hanno centro in un estremo di un segmento e passano per l'altro estremo del segmento.

3.4 Costruzioni geometriche

Come usare intersezioni tra oggetti.

Lo strumento base della geometria greca era lo spago:

- tenendo teso un pezzo di corda si poteva rappresentare un segmento allungabile a piacere;
- tenendo fisso un estremo e facendo ruotare l'altro, si poteva rappresentare una circonferenza.

Con questo strumento hanno costruito la geometria euclidea e risolto innumerevoli problemi.

3.4.1 Strumenti

In questo capitolo utilizzeremo i seguenti strumenti di Pyig:

- `Point(x, y)` crea un punto con date coordinate.
- `Line(p0, p1)` crea una retta passante per p0 e p1.
- `Segment(p0, p1)` crea un segmento di estremi p0 e p1.
- `Intersection(oggetto_0, oggetto_1, [which])` crea un punto di intersezione tra due oggetti.
- `Polygon((punto0, punto1, punto2, ...))` crea un poligono dati i vertici. tra due oggetti.

3.4.2 Problema

Crea un piano e disegna:

- nel primo quadrante: due punti e il triangolo equilatero costruito su quei due punti;
- nel secondo quadrante: un segmento e l'asse di quel segmento;
- nel terzo quadrante: un angolo e la bisettrice di quell'angolo;
- nel quarto quadrante: due punti e il quadrato costruito su quei due punti.

Soluzione guidata

1. Crea un nuovo programma e salvarlo con il nome: `gi03Costruzioni.py`. Per creare un nuovo programma: vedi la soluzione guidata del primo capitolo.
2. Scrivi un'intestazione adeguata.
3. Scrivi lo scheletro del programma.
4. Ora scriviamo dei commenti che indicano come intendiamo risolvere il problema:

```
# Disegno tre punti disposti accuratamente
# Disegno il poligono che passa per i tre punti
```

5. Risolviamo la prima parte del problema ottenendo un programma principale simile a questo:

```
# programma principale
ip = ig.InteractivePlane()

# Disegno tre punti disposti accuratamente
p_0 = ig.Point(1, 2, width=6)
p_1 = ig.Point(11, 2, width=6)
p_2 = ig.Point(6, 10.66, width=6)
# Disegno il poligono che passa per i tre punti
triequi = ig.Polygon((p_0, p_1, p_2),
                     width=5, color='green', intcolor='gold')
```

   Osservate che il costruttore di `Poligon` vuole un primo argomento formato da una sequenza di punti per cui i vertici del poligono devono essere racchiusi tra parentesi.

6. Proviamo il programma e controlliamo che rispetti le specifiche richieste dal problema. Cosa succede quando muovo i punti base?

   Accidenti, il triangolo è equilatero all'inizio, ma non lo è più quando sposto uno dei punti base.

   Dobbiamo affrontare il problema in un altro modo. Il terzo vertice va costruito partendo dai primi due:

```
# Disegno due punti in una posizione qualunque
# Disegno le due circonferenze che hanno centro in un punto e
# passano per l'altro
# Trovo un'intersezione delle due circonferenze
# Disegno il poligono che ha per vertici i due punti e l'intersezione
```

7. Dovremmo ottenere un programma che assomiglia a questo:

```
# programma principale
ip = ig.InteractivePlane()

# Disegno due punti in una posizione qualunque
p_0 = ig.Point(1, 2, width=6)
p_1 = ig.Point(11, 2, width=6)
# Disegno le due circonferenze che hanno centro in un punto e
# passano per l'altro
c_0 = ig.Circle(p_0, p_1, width=1)
c_1 = ig.Circle(p_1, p_0, width=1)
# Trovo un'intersezione delle due circonferenze
p_2 = ig.Intersection(c_0, c_1, +1, width=6)
# Disegno il poligono che ha per vertici i due punti e l'intersezione
triequi = ig.Polygon((p_0, p_1, p_2),
                     width=5, color='green', intcolor='gold')
```

   Osservate che è buona norma tenere le linee di costruzione più sottili rispetto alle altre, o addirittura renderle invisibili `visible=False`.

   Si può osservare che questa volta i punti liberi sono solo due, il terzo vertice è vincolato alla posizione di questi due dalla nostra costruzione. Ora, se muoviamo i punti base il nostro triangolo cambia posizione e dimensioni, ma resta sempre un triangolo equilatero come era richiesto dal problema.

8. A questo punto cerca su un libro di disegno su internet come risolvere gli altri tre problemi. Risolvili con riga e compasso, poi con `pyig`. Di seguito riporto le tre tracce di soluzione.

9. Asse di un segmento:

```
# Disegno due punti
# Disegno il segmento
# Disegno le due circ. che hanno centro in un estremo e passano per l'altro
# Chiamo i_0 e i_1 le due intersezioni delle circonferenze
# L'asse è la retta passante per i_0 e i_1
```

10. Bisettrice di un angolo:

```
# Disegno tre punti: p_0, vertice, p_1
# Disegno i due lati dell'angolo: r_0 e r_1
# Disegno una circ. che ha centro nel vertice e passa per p_0
# Chiamo i_1 l'intersezione della circonferenze con il lato r_1
# Disegno le circonferenze di centro p_0 e i_1 passanti per il vertice
# Chiamo i_2 l'intersezione delle due circonferenze
# La retta vertice - i_2 è la bisettrice cercata
```

11. Quadrato dati due vertici consecutivi:

```
# Disegno due punti: p_0, p_1
# c_0 è la circ. che ha centro in p_0 e passa per p_1
# c_1 è la circ. che ha centro in p_1 e passa per p_0
# i_0 è l'intersezione di queste due circonferenze c_0 e c_1
```

```
    # c_2 è la circ. che ha centro in i_0 e passa per p_0
    # i_1 è l'intersezione delle circonferenze c_0 e c_2
    # c_3 è la circonferenza di centro i_1 passante per p_0
    # i_2 è l'intersezione delle circonferenze c_3 e c_2
    # r_0 è la retta passante per p_0 e i_2
    # p_3 è l'intersezione della retta r_0 con la circonferenza c_0
    # c_4 è la circonferenza di centro i_3 passante per p_0
    # p_2 è l'intersezione della circonferenza c_4 con la circonferenza c_1
    # Il quadrato cercato e il poligono di vertici: (p_0, p_1, p_2, p_3)
```

Quando il programma e complicato, come in quest'ultimo caso è importante eseguire il programma ogni volta che si aggiunge un'istruzione in modo da individuare immediatamente eventuali errori sia sintattici sia logici.

12. Completiamo il programma per risolvere anche la terza parte del problema.

Riassumendo

- Polygon permette di disegnare un poligono data una sequenza di punti. La sintassi del costruttore di `Polygon` è:

```
Polygon(sequenza di punti [, intcolor=white]
              [, visible=True] [, color='blue'] [, width=3] [, name=''])
```

- Per affrontare problemi complicati: prima pianifica la soluzione descrivendola per mezzo di commenti, poi scrivi le istruzioni per risolvere il problema eseguendo il programma ad ogni modifica.
- Nei libri d disegno, o in internet, si possono trovare molte costruzioni geometriche che si possono realizzare con rette, circonferenze e intersezioni.

Prova tu

1. Disegna un quadrato dati due vertici opposti.
2. Disegna un esagono regolare dati due vertici consecutivi.
3. Disegna un esagono regolare dato il centro e un vertice.
4. Disegna un pentagono regolare dati due vertici consecutivi.
5. Disegna un parallelogramma dati tre vertici consecutivi.

3.5 Strumenti di uso comune

Quali altri oggetti abbiamo a disposizione.

Nel paragrafo precedente abbiamo visto come realizzare oggetti nuovi come assi, bisettrici, triangoli, quadrati, ... Ma se ho bisogno di vari assi per realizzare una costruzione complessa, non è comodo per ognuno di questi ripetere tutta la costruzione. Alcuni oggetti di uso comune sono già prefabbricati e vengono messi a disposizione dalla libreria pyig, basta chiamarli.

Nei prossimi paragrafi riporto quelli di uso più comune, l'elenco completo si trova nel manuale di Pygraph che è stato scaricato assieme alle librerie.

3.5.1 `InteractivePlane`

`InteractivePlane` Crea il piano interattivo nel fare questa operazione si possono decidere alcune caratteristiche.

Sintassi

```
<nome_variabile> = InteractivePlane([<parametri>])
```

Osservazioni

Il costruttore presenta molti parametri tutti con un valore predefinito. Nel momento in cui si crea un piano cartesiano si possono quindi decidere le sue caratteristiche. Vediamole in dettaglio:

- titolo della finestra, valore predefinito: "Interactive geometry";
- dimensione, valori predefiniti: larghezza=600, altezza=600;
- scala di rappresentazione, valori predefiniti: una unità = 20 pixel;
- posizione dell'origine, valore predefinito: il centro della finestra;
- rappresentazione degli assi cartesiani, valore predefinito: `True`;
- rappresentazione di una griglia di punti, valore predefinito: `True`;
- colore degli assi valore predefinito: '#808080' (grigio).
- colore della griglia valore predefinito: '#808080'.
- riferimento alla finestra che contiene il piano cartesiano, valore predefinito: `None`.

Poiché tutti i parametri hanno un valore predefinito, possiamo creare un oggetto della classe `InteractivePlane` senza specificare alcun argomento: verranno usati tutti i valori predefiniti. Oppure possiamo specificare per nome gli argomenti che vogliamo siano diversi dal comportamento predefinito, si vedano di seguito alcuni esempi.

Esempio

Si vedano tutti gli esempi seguenti.

3.5.2 `Point`

Scopo

Crea un *punto libero* date le coordinate della sua posizione iniziale.

Questo oggetto è la base di ogni costruzione; dai punti liberi dipendono, direttamente o indirettamente, gli altri oggetti grafici.

Quando il puntatore del mouse si sovrappone ad un punto libero questo cambia colore. Trascinando un punto libero, con il mouse, tutti gli oggetti che dipendono da lui, verranno modificati.

`Point` essendo un oggetto che può essere trascinato con il mouse ha un colore predefinito diverso da quello degli altri oggetti.

Sintassi

```
Point(x, y[, visible][, color][, width][, name])
```

Nota: Spesso nella pratica è necessario assegare l'oggetto creato ad un identificatore in modo da poter fare riferimento ad un oggetto nella costruzione di altri oggetti:

```
<identificatore> = Point(x, y[, visible][, color][, width][, name])
```

Si vedano gli esempi seguenti.

Osservazioni

- `x` e `y` sono due numeri, `x` è l'ascissa e `y` l'ordinata del punto.
- Per quanto riguarda i parametri non obbligatori si veda quanto scritto nel paragrafo relativo agli attributi degli oggetti visibili.

Nota: Nel resto del manuale riporterò solo gli argomenti obbligatori, è sottinteso che tutti gli oggetti che possono essere visualizzati hanno anche i parametri: `visible`, `color`, `width`, `name`.

Esempio

Funzione definita in N ad andamento casuale.

```
import random
ip = InteractivePlane('Point')
y = 0
for x in range(-14, 14):
    y += random.randrange(-1, 2)
    Point(x, y, color='red')
```

Attributi degli oggetti geometrici

Scopo

`Point`, come tutti gli oggetti geometrici ha degli attributi che possono essere determinati nel momento della creazione dell'oggetto stesso o in seguito. Questi attributi definiscono alcune caratteristiche degli oggetti che possono essere visualizzati.

- `visible` stabilisce se l'oggetto sarà visibili o invisibile;
- `color` imposta il colore dell'oggetto;
- `width` imposta la larghezza dell'oggetto.
- `name` imposta il nome dell'oggetto.

Sintassi

```
<oggetto>.visible = v
<oggetto>.color = c
<oggetto>.width = w
<oggetto>.name = s
```

Osservazioni

- `v` è un valore booleano, può essere True o False.
- `w` è un numero che indica la larghezza in pixel.
- `c` può essere:
  - una stringa nel formato: "#rrggbb" dove rr, gg e bb sono numeri esadecimali di due cifre che rappresentano rispettivamente le componenti rossa, verde, e blu del colore;
  - Una stringa contenente il nome di un colore;
  - Una terna di numeri nell'intervallo 0-1 rappresentanti le componenti rossa verde e blu.
- `s` è una stringa

Esempio

Disegna tre punti: uno con i valori di default, uno con colore dimensione e nome definiti quando viene creato, uno con valori cambiati dopo essere stato cerato.

```
ip = InteractivePlane('attributi')
a = Point(-5, 3)
b = Point(2, 3, color='indian red', width=8, name='B')
c = Point(9, 3)
c.color = 'dark orange'
c.width = 8
c.name = 'C'
```

Metodi degli oggetti geometrici

Scopo

Tutti gli oggetti geometrici hanno anche dei metodi che danno come risultato alcune informazioni relative all'oggetto stesso.

- `xcoord` l'ascissa;
- `ycoord` l'ordinata;
- `coords` le coordinate.

Sintassi

```
<oggetto>.xcoord()
<oggetto>.ycoord()
<oggetto>.coords()
```

Osservazioni

Non richiedono argomenti e restituiscono un particolare oggetto che può essere utilizzato all'interno di un testo variabile.

Esempio

Scrivi ascissa, ordinata e posizione di un punto.

```
ip = InteractivePlane('coords, xcoord, ycoord')
a = Point(-5, 8, name='A')
VarText(-5, -1, 'ascissa di A: {0}', a.xcoord())
VarText(-5, -2, 'ordinata di A: {0}', a.ycoord())
VarText(-5, -3, 'posizione di A: {0}', a.coords())
```

3.5.3 `Segment`

Scopo

Crea un segmento dati i due estremi, i due estremi sono *punti*.

Sintassi

```
<identificatore> = Segment(point0, point1)
```

Osservazioni

`point0` e `point1` sono due punti.

Esempio

Disegna un triangolo con i lati colorati in modo differente.

```
ip = InteractivePlane('Segment')
# creo i 3 vertici
v0 = Point(-4, -3, width=5)
v1 = Point( 5, -1, width=5)
v2 = Point( 2,  6, width=5)
# creo i 3 lati
l0 = Segment(v0, v1, color='steel blue')
l1 = Segment(v1, v2, color='sea green')
l2 = Segment(v2, v0, color='saddle brown')
```

`length`

Scopo

È il metodo della classe `Segment` che restituisce un oggetto `data` contenete la lunghezza del segmento stesso.

Sintassi

```
<obj>.length()
```

Osservazioni

La lunghezza è la distanza tra `point0` e `point1`.

Esempio

Disegna un segmento e scrivi la sua lunghezza.

```
ip = InteractivePlane('length')
p0 = Point(-4, 7, width=5, name='A')
p1 = Point(8, 10, width=5, name='B')
seg = Segment(p0, p1)
VarText(-5, -5, 'lunghezza di AB = {0}', seg.length())
```

3.5.4 MidPoints

Scopo

Crea il punto medio tra due punti.

Sintassi

```
MidPoints(point0, point1)
```

Osservazioni

`point0` e `point1` sono due punti.

Esempio

Punto medio tra due punti.

```
ip = InteractivePlane('MidPoints')
# creo due punti
p0 = Point(-2, -5)
p1 = Point(4, 7)
# cambio i loro attributi
p0.color = "#00a600"
p0.width = 5
p1.color = "#006a00"
p1.width = 5
# creao il punto medio tra p0 e p1
m = MidPoints(p0, p1, name='M')
# cambio gli attributi di m
m.color = "#f0f000"
m.width = 10
```

3.5.5 MidPoint

Scopo

Crea il punto medio di un segmento

Sintassi

```
MidPoint(segment)
```

Osservazioni

`segment` è un oggetto che ha un `point0` e un `point1`.

Esempio

Punto medio di un segmento.

```
ip = InteractivePlane('MidPoint')
# creo un segmento
s=Segment(Point(-2, -1, color="#a60000", width=5),
          Point(5, 7, color="#6a0000", width=5),
          color="#a0a0a0")
# creo il suo punto medio
MidPoint(s, color="#6f6f00", width=10, name='M')
```

3.5.6 `Line`

Scopo

Crea una retta per due punti.

Sintassi

```
Line(point0, point1)
```

Osservazioni

`point0` e `point1` sono, indovina un po', due punti.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Triangolo delimitato da rette.

```
ip = InteractivePlane('Line')
# creo i 3 punti
a=Point(0, 0)
b=Point(1, 5)
c=Point(5, 1)
# creo i 3 lati
Line(a, b, color="#dead34")
Line(b, c, color="#dead34")
Line(c, a, color="#dead34")
```

3.5.7 Ray

Scopo

Traccia una semiretta con l'origine in un punto e passante per un altro punto.

Sintassi

```
Ray(point0, point1)
```

Osservazioni

`point0` è l'origine della semiretta che passa per `point1`.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Triangolo delimitato da semirette.

```
ip = InteractivePlane('Ray')
# creo i 3 punti
a=Point(0, 0)
b=Point(1, 5)
c=Point(5, 1)
# creo i 3 lati
Ray(a, b, color="#de34ad")
Ray(b, c, color="#de34ad")
Ray(c, a, color="#de34ad")
```

3.5.8 Orthogonal

Scopo

Crea la retta perpendicolare ad una retta data passante per un punto.

Sintassi

```
Orthogonal(line, point)
```

Osservazioni

`line` è la retta alla quale si costruisce la perpendicolare passante per `point`.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Disegna la perpendicolare ad una retta data passante per un punto.

```
ip = InteractivePlane('Orthogonal')
retta = Line(Point(-4, -1, width=5),
             Point(6, 2, width=5),
             width=3, color='DarkOrange1', name='r')
punto = Point(-3, 5, width=5, name='P')
Orthogonal(retta, punto)
```

3.5.9 Parallel

Scopo

Crea la retta parallela ad una retta data passante per un punto.

Sintassi

```
Parallel(line, point)
```

Osservazioni

`line` è la retta alla quale si costruisce la parallela passante per `point`.

Vedi anche i metodi delle classi *linea* presentati nella classe `Segment`.

Esempio

Disegna la parallela ad una retta data passante per un punto.

```
ip = InteractivePlane('Parallel')
retta = Line(Point(-4, -1, width=5),
             Point(6, 2, width=5),
             width=3, color='DarkOrange1', name='r')
punto = Point(-3, 5, width=5, name='P')
Parallel(retta, punto)
```

3.5.10 Polygon

Scopo

Crea un poligono data una sequenza di vertici.

Sintassi

```
Polygon(points)
```

Osservazioni

`points` è una sequenza di punti, può essere una lista (delimitata da parentesi quadre) o una tupla (delimitata da parentesi tonde).

Esempio

Disegna un poligono date le coordinate dei vertici.

```
ip = InteractivePlane('24: Polygon')
# Lista di coordinate
coords = ((-8, -3), (-6, -2), (-5, -2), (-4, 2), (-2, 3), (0, 4),
          (2, 3), (4, 2), (5, -2), (6, -2), (8, -3))
# Costruzione di una lista di punti partendo da una lista di coordinate:
# listcompreension
ip.defwidth = 5
points = [Point(x, y) for x,y in coords]
Polygon(points, color='HotPink3')
```

`perimeter` e `surface`

Scopo

Sono metodi presenti in tutte le classi *figura*, restituiscono la lunghezza del contorno e l'area della superficie dell'oggetto.

Sintassi

```
<figura>.perimeter()
<figura>.surface()
```

Osservazioni

Sono metodi degli oggetti che sono *figure piane* e non richiede argomenti.

Esempio

Scrive alcune informazioni relative a un poligono.

```
poli = Polygon((Point(-7, -3, width=5, name="A"),
                Point(5, -5, width=5, name="B"),
                Point(-3, 8, width=5, name="C")),
                width=4, color='magenta', intcolor='olive drab')
VarText(-3, -6, "perimetro={0}", poli.perimeter(), color='magenta')
VarText(-3, -7, "area={0}", poli.surface(), color='olive drab')
```

3.5.11 `Circle`

Scopo

Circonferenza dato il centro e un punto o il centro e il raggio (un segmento).

Sintassi

```
Circle(center, point)
Circle(center, segment)
```

Osservazioni

`center` è il centro della circonferenza passante per `point` o di raggio segment.

Vedi anche i metodi delle classi *figure piane* presentati nella classe `Polygon`.

Esempio

Circonferenze con centro nell'origine.

```
ip = InteractivePlane('Circle(Point, Point)')
origine = Point(0, 0, visible=False, name="O")
p0 = Point(-7, -3, width=5, name="P")
Circle(origine, p0, color="#c0c0de", width=4)
raggio = Segment(Point(-7, 9, width=5, name="A"),
                 Point(-4, 9, width=5, name="B"))
Circle(origine, raggio, color="#c0c0de", width=4)
```

3.5.12 Intersection

Scopo

Crea il punto di intersezione tra due oggetti.

Sintassi

```
Intersection(obj0, obj1)
Intersection(obj0, obj1, which)
```

Osservazioni

`obj0` e `obj1` possono essere rette o circonferenze. Se uno dei due oggetti è una circonferenza è necessario specificare quale delle due intersezioni verrà restituita indicando come terzo parametro `+1` o `-1`.

Esempio

Disegna una circonferenza tangente a una retta.

```
ip = InteractivePlane('Intersection line line')
# Disegno retta e punto
retta = Line(Point(-4, -1, width=5),
             Point(6, 2, width=5),
             width=3, color='DarkOrange1', name='r')
punto = Point(-3, 5, width=5, name='P')
# trovo il punto di tangenza
perpendicolare = Orthogonal(retta, punto, width=1)
p_tang = Intersection(retta, perpendicolare, width=5)
# disegno la circonferenza
Circle(punto, p_tang, width=4, color='IndianRed')
```

Disegna il simmetrico di un punto rispetto ad una retta.

```
ip = InteractivePlane('Intersection line circle')
# disegno l'asse di simmetria e il punto
asse = Line(Point(-4, -11, width=5),
            Point(-2, 12, width=5),
            width=3, color='DarkOrange1', name='r')
punto = Point(-7, 3, width=5, name='P')
# disegno la perpendicolare all'asse passante per il punto
perp = Orthogonal(asse, punto, width=1)
# trovo l'intersezione tra la perpendicolare e l'asse
piede = Intersection(perp, asse)
# disegno la circonferenza di centro piede e passante per punto
circ = Circle(piede, punto, width=1)
# trovo il simmetrico di punto rispetto a asse
Intersection(perp, circ, -1, width=5, color='DebianRed', name="P'")
```

Disegna un triangolo equilatero.

```
ip = InteractivePlane('Intersection circle circle')
# Disegno i due primi vertici
```

```
v0=Point(-2, -1, width=5, name='A')
v1=Point(3, 2, width=5, name='B')
# Disegno le due circonferenze di centro p0 e p1 e passanti per p1 e p0
c0=Circle(v0, v1, width=1)
c1=Circle(v1, v0, width=1)
# terzo vertice: intersezione delle due circonferenze
v2=Intersection(c0, c1, 1, width=5, name='C')
# triangolo per i 3 punti
Polygon((v0, v1, v2), width=4, color='DarkSeaGreen4')
```

3.5.13 Text

Scopo

Crea un testo posizionato in un punto del piano.

Sintassi

```
Text(x, y, text[, iplane=None])
```

Osservazioni

- `x` e `y` sono due numeri interi o razionali relativi `x` è l'ascissa e `y` l'ordinata del punto.
- `text` è la stringa che verrà visualizzata.
- Se sono presenti più piani interattivi, si può specificare l'argomento `iplane` per indicare in quale di questi la scritta deve essere visualizzata.

Esempio

Scrive un titolo in due finestre grafiche.

```
ip0 = InteractivePlane('Text pale green', w=400, h=200)
ip1 = InteractivePlane('Text blue violet', w=400, h=200)
Text(-2, 2, "Prove di testo blue violet",
     color='blue violet', width=20)
Text(-2, 2, "Prove di testo pale green",
     color='pale green', width=20, iplane=ip0)
```

3.5.14 VarText

Scopo

Crea un testo variabile. Il testo contiene dei "segnaposto" che verranno sostituiti con i valori prodotti dai dati presenti nel parametro variables.

Sintassi

```
VarText(x, y, text, variables[, iplane=None])
```

Osservazioni

- `x` e `y` sono due numeri interi o razionali relativi `x` è l'ascissa e `y` l'ordinata del punto.
- `text` è la stringa che contiene la parte costante e i segnaposto.
- In genere i *segnaposto* saranno nella forma: "{0}" che indica a Python di convertire in stringa il risultato prodotto dal dato.
- `variables` è un dato o una tupla di dati.
- Se sono presenti più piani interattivi, si può specificare l'argomento `iplane` per indicare in quale di questi la scritta deve essere visualizzata.

Esempio

Un testo che riporta la posizione dei un punto.

```
ip = InteractivePlane('VarText')
p0 = Point(7, 3, color='green', width=10, name='A')
VarText(-4, -3, "Posizione del punto A: ({0}; {1})",
                (p0.xcoord(), p0.ycoord()),
                color='green', width=10)
```

3.5.15 PointOn

Scopo

Punto disegnato su un oggetto in una posizione fissa.

Sintassi

```
PointOn(obj, parameter)
```

Osservazioni

L'oggetto deve essere una linea o una retta o una circonferenza, `parameter` è un numero che individua una precisa posizione sull'oggetto. Sia le rette sia le circonferenze hanno una loro metrica che è legata ai punti base dell'oggetto. Su una retta una semiretta o un segmento `point0` corrisponde al parametro 0 mentre `point1` corrisponde al parametro 1. Nelle circoferenze il punto di base della circonferenza stessa corrisponde al parametro 0 l'intera circonferenza vale 2. Il punto creato con `PointOn` non può essere trascinato con il mouse.

Esempio

Disegna il simmetrico di un punto rispetto ad una retta.

```
ip = InteractivePlane('PointOn')
# disegno l'asse di simmetria e il punto
asse = Line(Point(-4, -11, width=5),
            Point(-2, 12, width=5),
            width=3, color='DarkOrange1', name='r')
punto = Point(-7, 3, width=5, name='P')
# disegno la perpendicolare all'asse passante per il punto
perp = Orthogonal(asse, punto, width=1)
# trovo il simmetrico di punto rispetto a asse
```

```
PointOn(perp, -1, width=5, color='DebianRed', name="P'")
Text(-5, -6, """P' è il simmetrico di P.""")
```

3.5.16 ConstrainedPoint

Scopo

Punto legato ad un oggetto.

Sintassi

```
ConstrainedPoint(obj, parameter)
```

Osservazioni

Per quanto riguarda `parameter`, valgono le osservazioni fatte per `PoinOn`. Questo punto però può essere trascinato con il mouse pur restando sempre sull'oggetto. Dato che può essere trascinato con il mouse ha un colore di default diverso da quello degli altri oggetti.

Esempio

Circonferenza e proiezioni sugli assi.

```
ip = InteractivePlane('ConstrainedPoint', sx=200)
# Circonferenza
origine = Point(0, 0, visible=False)
unix = Point(1, 0, visible=False)
uniy = Point(0, 1, visible=False)
circ = Circle(origine, unix, color="gray10")
# Punto sulla circonferenza
cursore = ConstrainedPoint(circ, 0.25, color='magenta', width=20)
# assi
assex = Line(origine, unix, visible=False)
assey = Line(origine, uniy, visible=False)
# proiezioni
py = Parallel(assey, cursore, visible=False)
hx = Intersection(assex, py, color='red', width=8)
px = Parallel(assex, cursore, visible=False)
hy = Intersection(assey, px, color='blue', width=8)
```

parameter

Scopo

I punti legati agli oggetti hanno un metodo che permette di ottenere il parametro.

Sintassi

```
<constrained point>.parameter()
```

Osservazioni

In `PointOn` il parametro è fissato nel momento della costruzione dell'oggetto. In `ConstrainedPoint` il parametro può essere variato trascinando il punto con il mouse.

Esempio

Scrivi i dati relativi a un punto collegato a un oggetto.

```
ip = InteractivePlane('parameter')
c0 = Circle(Point(-6, 6, width=6), Point(-1, 5, width=6))
c1 = Circle(Point(6, 6, width=6), Point(1, 5, width=6))
a = PointOn(c0, 0.5, name='A')
b = ConstrainedPoint(c1, 0.5, name='B')
VarText(-5, -1, 'ascissa di A: {0}', a.xcoord())
VarText(-5, -2, 'ordinata di A: {0}', a.ycoord())
VarText(-5, -3, 'posizione di A: {0}', a.coords())
VarText(-5, -4, 'parametro di A: {0}', a.parameter())
VarText(5, -1, 'ascissa di B: {0}', b.xcoord())
VarText(5, -2, 'ordinata di B: {0}', b.ycoord())
VarText(5, -3, 'posizione di B: {0}', b.coords())
VarText(5, -4, 'parametro di B: {0}', b.parameter())
```

3.5.17 `Angle`

Scopo

Angolo dati tre punti o due punti e un altro angolo. Il secondo punto rappresenta il vertice. Il verso di costruzione dell'angolo è quello antiorario.

Sintassi

```
Angle(point0, vertex, point1[, sides])
Angle(point0, vertex, angle[, sides])
```

Osservazioni

L'argomento `sides` può valere:

- `True` (o `(0, 1)`): vengono disegnati i lati;
- `0`: viene disegnato il lato 0;
- `1`: viene disegnato il lato 1;

`Angle` fornisce i seguenti metodi dal significato piuttosto evidente:

```
* ``extent``: ampiezza dell'angolo;
* ``bisector``: bisettrice;
```

Esempio

Disegna un angolo e un angolo con i lati.

```
ip = InteractivePlane('Angle(Point, Point, Point)')
ip.defwidth = 5
a = Point(-2, 4, color="#40c040", name="A")
```

```
b = Point(-5, -2, color="#40c040", name="B")
c = Point(-8, 6, color="#40c040", name="C")
d = Point(8, 6, color="#40c040", name="D")
e = Point(5, -2, color="#40c040", name="E")
f = Point(2, 4, color="#40c040", name="F")
# angolo senza i lati
Angle(a, b, c, color="#40c040")
# angolo con i lati
Angle(d, e, f, color="#c04040", sides=True)
```

Somma di due angoli.

```
ip = InteractivePlane('Angle(Point, Point, Angle)')
# i 2 angoli di partenza
a = Angle(Point(-3, 7, width=6),
          Point(-7, 5, width=6),
          Point(-6, 8, width=6),
          sides=(0, 1), color="#f09000", name='alfa')
b = Angle(Point(9, 2, width=6),
          Point(2, 3, width=6),
          Point(6, 4, width=6),
          sides=(0, 1), color="#0090f0", name='beta')
# Punti di base dell'angolo somma di a b
v = Point(-11, -8, width=6)
p0 = Point(3, -10, width=6)
# la somma degli angoli
b1 = Angle(p0, v, b, (0, 1), color="#0090f0")
p1 = b1.point1()
a1 = Angle(p1, v, a, sides=True, color="#f09000")
Text(-4, -12, "Somma di due angoli")
```

3.5.18 `Bisector`

Scopo

Retta bisettrice di un angolo.

Sintassi

```
Bisector(angle)
```

Osservazioni

Vedi `Ray`.

Esempio

Disegna l'incentro di un triangolo.

```
ip = InteractivePlane('Bisector')
# I tre vertici del triangolo
a=Point(-7, -3, color="#40c040", width=5, name="A")
```

```
b=Point(5, -5, color="#40c040", width=5, name="B")
c=Point(-3, 8, color="#40c040", width=5, name="C")
# Il triangolo
Polygon((a, b, c))
# Due angoli del triangolo
cba=Angle(c, b, a)
bac=Angle(b, a, c)
# Le bisettrici dei due angoli
b1=Bisector(cba, color="#a0c040")
b2=Bisector(bac, color="#a0c040")
# L'incentro
Intersection(b1, b2, color="#c040c0", width=5, name="I")
```

3.5.19 `Calc`

Scopo

Dato che contiene il risultato di un calcolo.

Sintassi

```
Calc(function, variables)
```

Osservazioni

- `function` è una funzione python, al momento del calcolo, alla funzione vengono passati come argomenti il contenuto di `variables`.
- `variables` è un oggetto `Data` o una tupla che contiene oggetti Data. Il risultato è memorizzato all'interno dell'oggetto Calc e può essere visualizzato con `VarText` o utilizzato per definire la posizione di un punto.

Esempio

Calcolla il quadrato di un lato e la somma dei quadrati degli altri due di un triangolo.

```
ip = InteractivePlane('Calc')
Circle(Point(2, 4), Point(-3, 4), width=1)
ip.defwidth = 5
a = Point(-3, 4, name="A")
b = Point(7, 4, name="B")
c = Point(-1, 8, name="C")
ab = Segment(a, b, color="#40c040")
bc = Segment(b, c, color="#c04040")
ca = Segment(c, a, color="#c04040")
q1 = Calc(lambda a: a*a, ab.length())
q2 = Calc(lambda a, b: a*a+b*b, (bc.length(), ca.length()))
VarText(-5, -5, "ab^2 = {0}", q1, color="#40c040")
VarText(-5, -6, "bc^2 + ca^2 = {0}", q2, color="#c04040")
```

Riassumendo

- In questo paragrafo sono stati presentati i seguenti oggetti.

  - `Angle` Angolo dati tre punti o due punti e un angolo, il secondo punto rappresenta il vertice. Il verso di costruzione dell'angolo è quello antiorario.
  - `Bisector` Retta bisettrice di un angolo.
  - `Circle` Circonferenza dato il centro e un punto o il centro e un raggio (un segmento).
  - `ConstrainedPoint` Punto legato ad un oggetto.
  - `Calc` Dato che contiene il risultato di un calcolo.
  - `InteractivePlane` Crea il piano cartesiano e inizializza gli attributi del *piano*.
  - `Intersection` Crea il punto di intersezione tra due rette.
  - `Line` Crea una retta per due punti.
  - `MidPoint` Crea il punto medio di un segmento
  - `MidPoints` Crea il punto medio tra due punti.
  - `Orthogonal` Crea la retta perpendicolare ad una retta data passante per un punto.
  - `Parallel` Crea la retta parallela ad una retta data passante per un punto.
  - `Point` Crea un *punto libero* date le coordinate della sua posizione iniziale.
  - `PointOn` Punto disegnato su un oggetto in una posizione fissa.
  - `Polygon` Crea un poligono data una sequenza di vertici.
  - `Ray` Traccia una semiretta con l'origine in un punto e passante per un altro punto.
  - `Segment` Crea un segmento dati i due estremi, i due estremi sono *punti*.
  - `Text` Crea un testo posizionato in un punto del piano.
  - `VarText` Crea un testo variabile. Il testo contiene dei "segnaposto" che verranno sostituiti con i valori prodotti dai dati presenti nel parametro variables.

- In questo paragrafo sono stati presentati i seguenti attributi.
  - `<oggetto_visibile>.color` Attributo degli oggetti geometrici: imposta il colore dell'oggetto;
  - `<oggetto_visibile>.name` Attributo degli oggetti geometrici: imposta il nome dell'oggetto.
  - `<oggetto_visibile>.visible` Attributo degli oggetti geometrici: stabilisce se l'oggetto sarà visibili o invisibile;
  - `<oggetto_visibile>.width` Attributi degli oggetti geometrici: imposta la larghezza dell'oggetto.
- In questo paragrafo sono stati presentati i seguenti metodi.
  - `<circonferenza>.radius` Metodo delle classi *circonferenza* che restituisce un oggetto `data` che contiene la lunghezza del raggio della circonferenza.

- `<figura>.perimeter` Metodo delle classi *figura* che restituisce un oggetto `data` contenete la lunghezza del contorno dell'oggetto.
- `<figura>.surface` Metodo delle classi *figura* che restituisce un oggetto `data` contenete l'area della superficie dell'oggetto.
- `<oggetto_visibile>.coords` Restituisce un dato che contiene le coordinate.
- `<oggetto_visibile>.xcoord` Metodo degli oggetti visualizzabili: restituisce un dato che contiene l'ascissa.
- `<oggetto_visibile>.ycoord` Metodo degli oggetti visualizzabili: restituisce un dato che contiene l'ordinata.
- `<punto_legato>.parameter` Metodo dei punti legati agli oggetti che restituisce un oggetto `data` contenete il parametro.
- `<segmento>.length` Metodo della classe `Segment` che restituisce un oggetto `data` contenete la lunghezza del segmento stesso.

Prova tu

1. Ricopia e modifica alcuni esempi del manuale.
2. Disegna un triangolo con evidenziati i punti medi dei lati.
3. Disegna un quadrato usando gli oggetti: `Orthogonal` e `Parallel`.
4. Disegna un esagono regolare dato il centro e un vertice.
5. Disegna un poligono regolare dato il centro, un vertice e il numero di lati.
6. Disegna un poligono regolare e tutte le sue diagonali.

3.6 Trasformazioni geometriche nel piano

Nei prossimi capitoli studieremo alcune trasformazioni geometriche nel piano.

Delle trasformazioni cercheremo di capire:

1. se cambiano la forma o le dimensioni delle figure che trasformano;
2. se esistono delle figure che non si modificano nella trasformazione, cioè se la trasformazione ha degli elementi uniti;
3. alcune trasformazioni particolari;
4. le equazioni della trasformazione.

Per esplorare le trasformazioni nel piano useremo i seguenti strumenti della geometria interattiva:

- `Point(x, y)` crea un punto con date coordinate.
- `Line(p0, p1)` crea una retta passante per `p0` e `p1`.

- `Parallel(retta, punto)` crea una retta parallela a `retta` passante per `punto`.
- `Orthogonal(retta, punto)` crea una retta perpendicolare a `retta` passante per `punto`.
- `PointOn(oggetto, parametro)` crea un punto fissato su oggetto nella posizione definita da parametro.
- `Segment(p0, p1)` crea un segmento di estremi `p0` e `p1`.
- `MidPoint(segmento)` crea il punto medio di segmento.
- `ConstrainedPoint(object, parameter)` crea un punto vincolato a oggetto nella posizione iniziale definita da parametro.
- `Polygon(vertici)` crea un poligono data una sequenza di punti.
- `Circle(centro, punto)` crea una circonferenza di centro `centro`, passante per `punto`.
- `<poligono>.vertices` contiene la lista dei vertici del poligono.
- `<segmento>.length()` restituisce la lunghezza di un segmento.
- `<oggetto>.coords()` restituisce le coordinate di oggetto.
- `VarText(x, y, stringa, variabili)` crea un testo variabile nella posizione `x`, `y`.

Se ci sono dei dubbi sul loro significato conviene dare un'occhiata alla parte sull'informatica o al manuale di `pygraph`.

3.7 Traslazione

In questo capitolo si affrontano i seguenti argomenti:

1. Cos'è una traslazione e quali sono le sue proprietà.
2. Cosa sono gli elementi uniti in una traslazione.
3. Cosa sono le traslazioni in un poligono.
4. Cosa dice l'algebra sulle traslazioni.

3.7.1 Definizione

Nella geometria euclidea, una traslazione è una trasformazione che sposta tutti i punti nella stessa direzione di una distanza fissa.

In altre parole, dato un vettore, diremo che un punto P' è il traslato del punto P se il segmento `PP'` ha la stessa direzione, lo stesso verso e la stessa lunghezza del vettore.

La funzione principale che realizzeremo è quella che, dato un punto e un vettore, costruisce il traslato del punto rispetto al vettore. Si dovrà poterla chiamare in questo modo:

```
p_1 = traslapunto(p_0, traslazione)
```

Ovviamente `p_0` e `traslazione` dovranno essere rispettivamente un punto e un vettore creati precedentemente. Dopo la chiamata, `p_1` conterrà il riferimento al traslato di `p_0` della quantità indicata da `vettore`. Un frammento completo di programma potrebbe essere:

```
# Creo il vettore traslazione
trasl = ig.Vector(ig.Point(-13, 10, width=6),
                  ig.Point(-4, 12, width=6), name='t')

# Punto A, il suo traslato
a_0 = ig.Point(-3, 9, width=6, name="A")
a_1 = traslapunto(a_0, trasl, width=6, name="A'")
```

La funzione `traslapunto(punto, traslazione)` dovrà:

1. Creare una retta invisibile parallela a `traslazione` passante per `punto`.
2. Creare su questa retta un punto fisso nella posizione +1.
3. Dare come risultato questo punto.

Una possibile soluzione:

```
def traslapunto(punto, traslazione, **kargs):
  """Restituisce il punto traslato di traslazione."""
 parallela = ig.Parallel(traslazione, punto, False)
  return ig.PointOn(parallela, +1, **kargs)
```

Avviato IDLE crea una nuova finestra (menu-File-New window) e la salviamo, in una nostra cartella, con il nome `trasla01_proprieta.py`. Inizia questo programma con un'intestazione adeguata: alcuni commenti che contengano la data, il tuo nome e un titolo (ad esempio: `Traslazioni: proprietà`).

Scrivi ora un programma che disegni un vettore, un punto e il suo traslato.

Il programma potrà assomigliare a questo:

```
# data
# autore
# Traslazioni: proprietà

# lettura delle librerie
import pyig as ig

# funzioni
def traslapunto(punto, traslazione, **kargs):
    """Restituisce il punto traslato di traslazione."""
    parallela = ig.Parallel(traslazione, punto, False)
    return ig.PointOn(parallela, +1, **kargs)

# programma principale
ip = ig.InteractivePlane()
```

```
# Creo il vettore traslazione
trasl = ig.Vector(ig.Point(-13, 10, width=6),
                  ig.Point(-4, 12, width=6), name='t')

# Punto A e il suo punto traslato e il vettore AA'
a_0 = ig.Point(-5, 6, width=6, name="A")
a_1 = traslapunto(a_0, trasl, width=6, name="A'")
v_a = ig.Vector(a_0, a_1, width=1)

# attivazione della finestra grafica
ip.mainloop()
```

Esegui il programma, muovi i punti base, il punto A' deve rimanere sempre il traslato di A secondo il vettore dato. Se tutto funziona sei pronto per esplorare le caratteristiche delle simmetrie assiali.

3.7.2 Proprietà

Crea il vettore AA', con spessore 1. Esegui il programma e muovi il punto A: cosa puoi dire del segmento AA'?

. . . . . .

Costruisci ora un nuovo punto B, il suo traslato B' e il vettore BB' (spessore 1).

Costruisci i segmenti AB e A'B' (di un colore diverso dagli altri oggetti realizzati). Visualizza le misure di AB e A'B' usando la classe VarText:

```
ab =ig.Segment(a_0, b_0, width=6, color='violet')
a1b1 =ig.Segment(a_1, b_1, width=6, color='violet')
ig.VarText(-7, -7, "AB = {}", ab.length())
ig.VarText(-7, -8, "A'B' = {}", ab.length())
```

Muovi i punti base, cosa osservi?

. . . . . .

Puoi formulare la congettura: A'B' è congruente ad AB e prova a dimostrarla.

. . . . . .

. . . . . .

. . . . . .

Costruisci un punto P vincolato al segmento AB e il suo traslato P':

```
p_0 = ig.ConstrainedPoint(ab, .3, width=6, color='olive drab', name="P")
p_1 = traslapunto(p_0, trasl, width=6, color='olive drab', name="P'")
```

Muovi il punto P, cosa osservi?

. . . . . .

Costruisci un nuovo punto `C` il suo simmetrico `C'`, costruisci il poligono `ABC` e il poligono `A'B'C'`. Cosa si può concludere circa i triangoli `ABC` e `A'B'C'`?

. . . . . .

Cosa puoi dire sull'orientamento dei vertici del triangolo `ABC` e del suo trasformato `A'B'C'`?

. . . . . .

Riassumendo

- La traslazione è una trasformazione geometrica che trasforma segmenti in segmenti congruenti, perciò è una *isometria*.
- La traslazione mantiene il verso dei poligoni.
- Se un punto appartiene ad un segmento, il suo traslato appartiene al traslato del segmento.
- Il programma completo:

```
# Traslazioni: proprietà

# lettura delle librerie
import pyig as ig

# funzioni
def traslapunto(punto, traslazione, **kargs):
    """Restituisce il punto traslato di traslazione."""
##    return punto + traslazione
    parallela = ig.Parallel(traslazione, punto, False)
    return ig.PointOn(parallela, +1, **kargs)

# Programma principale
ip = ig.InteractivePlane()

# Creo il vettore traslazione
trasl = ig.Vector(ig.Point(-13, 10, width=6),
                  ig.Point(-4, 12, width=6), name='t')

# Punto A e il suo punto traslato
a_0 = ig.Point(-5, 6, width=6, name="A")
a_1 = traslapunto(a_0, trasl, width=6, name="A'")
v_a = ig.Vector(a_0, a_1, width=1)

# Punto B, B', il vettore BB' e il punto medio
b_0 = ig.Point(-7, 0, width=6, name="B")
b_1 = traslapunto(b_0, trasl, width=6, name="A'")
v_b = ig.Vector(b_0, b_1, width=1)

# Segmento AB e A'B'
ab =ig.Segment(a_0, b_0, width=6, color='violet')
a1b1 =ig.Segment(a_1, b_1, width=6, color='violet')
ig.VarText(-7, -7, "AB = {}", ab.length())
ig.VarText(-7, -8, "A'B' = {}", ab.length())
```

```
    # P vincolato alla retta AB
    p_0 = ig.ConstrainedPoint(ab, .3, width=6, color='olive drab', name="P")
    p_1 = traslapunto(p_0, trasl, width=6, color='olive drab', name="P'")

    # Punto C, C' e i triangoli ABC e A'B'C'
    c_0 = ig.Point(1, 5, width=6, name="B")
    c_1 = traslapunto(c_0, trasl, width=6, name="A'")
    ig.Polygon((a_0, b_0, c_0), width=4, color='violet', intcolor='gold')
    ig.Polygon((a_1, b_1, c_1), width=4, color='violet', intcolor='gold')

    # attivazione della finestra grafica
    ip.mainloop()
```

3.7.3 Elementi uniti

Un elemento unito è un oggetto geometrico che viene trasformato in se stesso da una trasformazione.

Avvia un nuovo programma e salvarlo con il nome: `trasla02_elementiuniti.py` e scrivi funzione `traslapunto(punto, traslazione, **kargs)` che restituisce il traslato di un punto. Nel programma principale crea un punto e il suo traslato. Il programma dovrebbe assomigliare a:

```
# Traslazioni: elementi uniti

# lettura delle librerie
import pyig as ig

# funzioni
def traslapunto(punto, traslazione, **kargs):
    """Restituisce il punto traslato di traslazione."""
    parallela = ig.Parallel(traslazione, punto, False)
    return ig.PointOn(parallela, +1, **kargs)

# Programma principale
ip = ig.InteractivePlane()

# Creo il vettore traslazione
trasl = ig.Vector(ig.Point(-13, 10, width=6),
                  ig.Point(-4, 12, width=6), name='t')

# Punto A e il suo traslato
a_0 = ig.Point(-5, 6, width=6, name="A")
a_1 = traslapunto(a_0, trasl, width=6, name="A'")

# attivazione della finestra grafica
ip.mainloop()
```

Esegui il programma, muovi i punti base, se tutto funziona puoi iniziare l'esplorazione degli elementi uniti della simmetria assiale.

Sono pochi gli elementi uniti in una traslazione, solo le rette parallele al vettore traslazione. Crea:

- una retta con uno spessore maggiore passante per `A` e parallela al vettore traslazione.
- una retta con uno spessore minore e di un altro colore passante per `A'` e parallela al vettore traslazione.

Qualunque sia la traslazione e qualunque sia il punto `A`, ottieni due rette sovrapposte: cioè `r'` coincide con `r`.

Riassumendo

- In una trasformazione un elemento si dice unito se viene trasformato in se stesso.
- In una traslazione, sono elementi uniti solo:
  - le rette . . . . . .

3.7.4 Equazioni delle traslazioni

Un vettore è completamente determinato dalla differenza delle coordinate tra il punto iniziale e il punto finale di un segmento orientato.

Avvia una nuova finestra di editor e salvarla con il nome: `trasla03_equazioni.py`. In questa finestra ricopia il seguente programma:

```
# Traslazioni: equazioni

# lettura delle librerie
import pyig as ig

# funzioni
def traslapunto(punto, traslazione, **kargs):
    """Restituisce il punto traslato di traslazione."""
##    return punto + traslazione
    parallela = ig.Parallel(traslazione, punto, False)
    return ig.PointOn(parallela, +1, **kargs)

# Programma principale
ip = ig.InteractivePlane()

# Creo il vettore traslazione
v = ig.Vector(ig.Point(0, 0, width=6),
              ig.Point(4, 3, width=6), name='t')

# Quattro punti
a_0 = ig.Point(-5, 6, width=6, name="A")
b_0 = ig.Point(3, 6, width=6, name="B")
c_0 = ig.Point(-6, -7, width=6, name="C")
```

```
d_0 = ig.Point(7, -4, width=6, name="D")

# Lista con quattro punti
punti = [a_0, b_0, c_0, d_0]

# Vettore v applicato a tutti i punti
for punto in punti:
    v_p = ig.Vector(punto, v)

# attivazione della finestra grafica
ip.mainloop()
```

Esegui il programma, correggi eventuali errori. Quanti vettori vedi?

Il programma produce complessivamente cinque segmenti orientati, ma questi rappresentano un solo vettore.

È un po' come le cinque frazioni seguenti:

$$\frac{9}{15}; \quad \frac{3}{5}; \quad \frac{18}{30}; \quad \frac{6}{10}; \quad \frac{30}{50};$$

rappresentano un solo numero razionale.

Nel programma principale crea un punto `P(5, 5)`, il suo traslato e aggiungi alcune istruzioni che visualizzino le componenti del vettore `v` e le coordinate del punto `P` e `P'`:

```
# Relazione tra componenti della traslazione e
# coordinate del punto traslato
p_0 = ig.Point(5, 5, width=6, name="P")
p_1 = traslapunto(a_0, v, width=6, name="P'")

ig.VarText(-7, -10, "v = {}", v.components())
ig.VarText(-7, -11, "P = {}", p_0.coords())
ig.VarText(-7, -12, "P' = {}", p_1.coords())
```

Modifica il vettore `v` e completa la seguente tabella lasciando fisso il punto `P(5, 5)`:

traslazione	simmetrico rispetto asse x
v (4; 3)	P'(. . . . . ; . . . . .)
v (1; -4)	P'(. . . . . ; . . . . .)
v (. . ; . .)	P'(x_p . . . ; y_p . . .)
v (a; b)	P'(. . . . . ; . . . . .)

Nella traslazione di componenti `(a, b)`: l'ascissa del generico punto `P'` traslato di `P` è . . . . . ; l'ordinata del generico punto `P'`, è . . . . . .

La traslazione si può tradurre nel sistema di equazioni: $\tau \begin{cases} x' = \\ y' = \end{cases}$

Riassumendo

- L'equazione della traslazione di vettore `v(a; b)` è:

$$\tau \begin{cases} x' = x + a \\ y' = y + b \end{cases}$$

Prova tu

Sul quaderno completa le seguenti frasi.

1. Una traslazione è
2. In una traslazione figure corrispondenti sono
3. In una traslazione sono unite
4. Le equazioni della traslazione di componenti `(a; b)` è:

3.8 Simmetria assiale

In questo capitolo si affrontano i seguenti argomenti:

1. Cos'è una simmetria assiale e quali sono le sue proprietà.
2. Cosa sono gli elementi uniti in una simmetria assiale.
3. Cosa sono gli assi di simmetria in un poligono.
4. Cosa dice l'algebra sulle simmetrie assiali.

3.8.1 Definizione

Una simmetria assiale di asse `asse` è una trasformazione che manda un punto `P` in un punto `P' appartenente alla retta perpendicolare all'asse di simmetria in modo tale che la distanza di ''P` dall'asse sia uguale alla distanza di `P'` dall'asse.

In altre parole, un punto P' è simmetrico del punto `P` rispetto alla retta `asse` se il segmento `PP'` è perpendicolare a `asse` e `asse` taglia a metà il segmento `PP'`.

La funzione principale che realizzeremo è quella che, dato un punto e una retta, costruisce il simmetrico del punto rispetto alla retta. Si dovrà poterla chiamare in questo modo:

```
p_1 = simmpunto(p_0, asse)
```

Ovviamente `p_0` e `asse` dovranno essere rispettivamente un punto e una retta creati precedentemente. Dopo la chiamata, `p_1` conterrà il riferimento al simmetrico di `p_0` rispetto a `asse`.

La funzione `simmpunto(punto, asse)` dovrà:

1. Creare una retta invisibile ortogonale a `asse` passante per `punto`.
2. Creare su questa retta un punto fisso nella posizione -1.
3. Dare come risultato questo punto.

Una possibile soluzione:

```
def simmpunto(punto, asse):
  """Restituisce il simmetrico di punto rispetto a asse."""
 perpendicolare = ig.Orthogonal(asse, punto, False)
  puntosimmetrico = ig.PointOn(perpendicolare, -1)
  return puntosimmetrico
```

La funzione proposta nel programma a fine capitolo è un po' più concisa e, in più, usa una particolare sintassi di Python che permette di passare un numero variabile di parametri definiti per chiave.

In questo modo si possono effettuare chiamate di questo tipo:

```
a_1 = simmpunto(a_0, asse, name="A'")
b_1 = simmpunto(a_0, asse, name="B'", color="navy")
c_1 = simmpunto(a_0, asse, name="C'", width=7)
```

Avviato IDLE creiamo una nuova finestra (menu-File-New window) e la salviamo, in una nostra cartella, con il nome `simass01_proprieta.py`. Iniziamo questo programma con un'intestazione adeguata: alcuni commenti che contengano la *data*, il nostro *nome* e un *titolo*.

Il programma potrà assomigliare a questo:

```
# 10/9/14
# Daniele Zambelli
# Simmetrie assiali

# lettura delle librerie
import pyig as ig

# funzioni
def simmpunto(punto, asse, **kags):
    """Restituisce il simmetrico di punto rispeto a asse."""
    perpendicolare = ig.Orthogonal(asse, punto, visible=False)
    return ig.PointOn(perpendicolare, -1, **kags)

# programma principale
ip = ig.InteractivePlane()

# Creo l'asse di simmetria
asse = ig.Line(ig.Point(-3, -12, width=6),
               ig.Point(2, 10, width=6), name='asse')

# Punto A, il suo punto simmetrico
a_0 = ig.Point(-3, 9, width=6, name="A")
a_1 = simmpunto(a_0, asse, width=6, name="A'")

# attivazione della finestra grafica
ip.mainloop()
```

Eseguiamo il programma, muoviamo i punti base, il punto `A'` deve rimanere sempre simmetrico di `A`. Se tutto funziona siamo pronti per esplorare le caratteristiche delle simmetrie assiali.

3.8.2 Proprietà

Crea il segmento AA′, con spessore 1, e costruisci il punto medio M. Esegui il programma e muovi il punto A: cosa puoi dire del segmento AA′ e del suo punto medio?

. . . . . .

Costruisci ora un nuovo punto B dalla stessa parte di A e il suo simmetrico B′ rispetto alla retta asse, costruisci il segmento BB′ (spessore 1) e il suo punto medio chiamandolo N. Puoi prevedere il comportamento di N?

. . . . . .

Costruisci i segmenti AB e A′B′ (di un colore diverso dagli altri oggetti realizzati). Visualizza le misure di AB e A′B′ usando la classe VarText:

```
ab =ig.Segment(a, b, width=6, color='violet')
a1b1 =ig.Segment(a1, b1, width=6, color='violet')
ig.VarText(-7, -7, "AB = {}", ab.length())
ig.VarText(-7, -8, "A'B' = {}", ab.length())
```

Muovi i punti base, cosa osservi?

. . . . . .

Puoi formulare la congettura: A′B′ è congruente ad AB. Aggiungi i due segmenti: MB e MB′ e prova a dimostrarla.

. . . . . .

. . . . . .

. . . . . .

Costruisci un punto P vincolato al segmento AB e il suo simmetrico P′:

```
p = ig.ConstrainedPoint(ab, .3, width=6, color='olive drab', name="P")
p1 = simmpunto(p, asse, width=6, color='olive drab', name="P'")
```

Muovi il punto P, cosa osservi?

. . . . . .

Costruisci un nuovo punto C dalla stessa parte di A e B rispetto a asse e il suo simmetrico C′, costruisci il poligono ABC, e il poligono A′B′C′. Cosa si può concludere circa i triangoli ABC e A′B′C′?

. . . . . .

Cosa puoi dire sull'orientamento dei vertici del triangolo ABC e del suo trasformato A′B′C′?

. . . . . .

Riassumendo

- La simmetria assiale è una trasformazione geometrica che trasforma segmenti in segmenti congruenti, perciò è una *isometria*.
- La simmetria assiale inverte il verso dei poligoni.

- Se un punto appartiene ad un segmento, il suo simmetrico appartiene al simmetrico del segmento.
- Il programma completo:

```
# Simmetrie assiali: proprietà

# lettura delle librerie
import pyig as ig

# funzioni
def simmpunto(punto, asse, **kags):
    """Restituisce il simmetrico di punto rispeto a asse."""
    perpendicolare = ig.Orthogonal(asse, punto, visible=False)
    return ig.PointOn(perpendicolare, -1, **kags)

# programma principale
ip = ig.InteractivePlane()

# Creo l'asse di simmetria
asse = ig.Line(ig.Point(-3, -12, width=6),
               ig.Point(2, 10, width=6), name='asse')

# Punto A, il suo simmetrico
a_0 = ig.Point(-3, 9, width=6, name="A")
a_1 = simmpunto(a_0, asse, width=6, name="A'")
# Il segmento AA' e il punto medio
sa =ig.Segment(a_0, a_1, width=1)
m = ig.MidPoint(sa, width=6, color='red', name="M")

# Punto B, il suo punto simmetrico
b_0 = ig.Point(-7, 3, width=6, name="B")
b_1 = simmpunto(b_0, asse, width=6, name="B'")
# Il segmento BB' e il punto medio
sb =ig.Segment(b_0, b_1, width=1)
n = ig.MidPoint(sb, width=6, color='red', name="N")

# Segmento AB e A'B'
ab =ig.Segment(a_0, b_0, width=6, color='violet')
a1b1 =ig.Segment(a_1, b_1, width=6, color='violet')
ig.VarText(-7, -7, "AB = {}", ab.length())
ig.VarText(-7, -8, "A'B' = {}", ab.length())
mb =ig.Segment(m, b_0, width=1)
mb1 =ig.Segment(m, b_1, width=1)

# P vincolato alla retta AB
p_0 = ig.ConstrainedPoint(ab, .3, width=6,
                          color='olive drab', name="P")
p_11 = simmpunto(p_0, asse, width=6,
                 color='olive drab', name="P'")

# Punto C, il suo punto simmetrico, i triangoli ABC e A'B'C'
```

```
    c_0 = ig.Point(-10, 5, width=6, name="B")
    c_1 = simmpunto(c_0, asse, width=6, name="B'")
    ig.Polygon((a_0, b_0, c_0),
               width=4, color='violet', intcolor='gold')
    ig.Polygon((a_1, b_1, c_1),
               width=4, color='violet', intcolor='gold')

    # attivazione della finestra grafica
    ip.mainloop()
```

3.8.3 Elementi uniti

Avvia un nuovo programma e salvarlo con il nome: `simmass02_elementiuniti.py` e scrivi funzione `simmpunto(punto, asse, **kags)` che restituisce il simmetrico di un punto rispetto a una retta. Nel programma principale crea tre punti e i loro simmetrici. Il programma dovrebbe assomigliare a:

```
# Simmetrie assiali: elementi uniti

# lettura delle librerie
import pyig as ig

# funzioni
def simmpunto(punto, asse, **kags):
    """Restituisce il simmetrico di punto rispeto a asse."""
    perpendicolare = ig.Orthogonal(asse, punto, visible=False)
    return ig.PointOn(perpendicolare, -1, **kags)

# programma principale
ip = ig.InteractivePlane()

# Creo l'asse di simmetria
asse = ig.Line(ig.Point(-3, -12, width=6),
               ig.Point(2, 10, width=6), name='asse')

# Punto A, B, C e i loro simmetrici A', B', C'
a_0 = ig.Point(-3, 9, width=6, name="A")
b_0 = ig.Point(-7, 3, width=6, name="B")
c_0 = ig.Point(-9, 6, width=6, name="C")
a_1 = simmpunto(a_0, asse, width=6, name="A'")
b_1 = simmpunto(b_0, asse, width=6, name="B'")
c_1 = simmpunto(c_0, asse, width=6, name="B'")

# attivazione della finestra grafica
ip.mainloop()
```

Esegui il programma, muovi i punti base, se tutto funziona puoi iniziare l'esplorazione degli elementi uniti della simmetria assiale.

Sposta uno dei punti sulla retta `asse`. Cosa osservi?

. . . . . .

In una trasformazione geometrica un punto viene detto *unito* se, trasformato, corrisponde a se stesso. Puoi concludere che:

. . . . . .

In generale, in una trasformazione geometrica, una figura viene detta *unita* quando è trasformata in se stessa (anche se non ogni suo punto è unito).

Un segmento che ha gli estremi su `asse` è . . . . . rispetto alla simmetria e è costituito da . . . . . .

Costruisci un triangolo `ABC` e il suo simmetrico `A'B'C'`. Muovi i punti `ABC` in modo che il triangolo simmetrico si sovrapponga al triangolo `A'B'C'`. Come deve essere il triangolo `ABC` per essere unito rispetto alla simmetria?

. . . . . .

Costruisci e descrivi altri elementi uniti rispetto alla simmetria.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Riassumendo

- In una trasformazione un elemento si dice unito se viene trasformato in se stesso.
- In una simmetria assiale sono elementi uniti:
  - i punti . . . . . .
  - i segmenti . . . . . .
  - le rette . . . . . .
  - le circonferenze . . . . . .
  - i triangoli . . . . . .
  - i poligoni . . . . . .

3.8.4 Poligoni simmetrici

Avvia un nuovo programma e salvarlo con il nome: `simass03_poligoni.py`. Scrivi la solita funzione `simmpunto`.

Scrivi una funzione che, dati `centro`, `vertice` e `numlati`, costruisca il poligono regolare. Lo schema potrebbe essere:

```
def polreg(centro, vertice, numlati, **kargs):
    """Restituisce un poligono regolare
       dati il centro un vertice e il numero di lati."""
    # crea la circonferenza su cui sono disposti i vertici non visibile
    # calcola la lunghezza dell'arco tra due vertici consecutivi
    # crea la lista dei vertici che contiene quello dato come argomento
    # aggiungi alla lista dei vertici tutti gli altri
    # restituisci il poligono costruito con questi vertici
```

Scrivi la funzione che, dati `poligono` e `asse`, costruisca il poligono simmetrico. Lo schema potrebbe essere:

```
def simmpoli(poligono, asse, **params):
    """Restituisce il simmetrico di un poligono rispetto a asse."""
    # crea una lista vuota che conterrà i vertici del poligono simmetrico
    # per ogni vertice del poligono originale, calcola il simmetrico e
    # aggiungilo alla lista dei vertici simmetrici
    # restituisci il poligono costruito con questi vertici
```

Nel programma principale crea:

- un piano interattivo;
- crea il punto O di coordinate (6, 3);
- l'asse passante per quel punto e il punto (6, 7);
- il triangolo equilatero di centro O e passante per (4, 3), usa la funzione `polreg`;
- il simmetrico del triangolo (usa la funzione `simmpoli`).

Una figura è simmetrica rispetto ad un *asse* se resta unita nella simmetria.

Agendo con il mouse, muovi la retta `asse` facendo in modo che il triangolo trasformato si sovrapponga al triangolo originale.

Sono tre e sono quelle in cui l'asse passa per . . . . . .

. . . . . .

Ripeti le operazioni precedenti disegnando un quadrato nel secondo quadrante, un pentagono regolare nel terzo e un esagono regolare nel quarto, sempre con un asse di simmetria passante per il centro. Cosa puoi osservare?

. . . . . .

. . . . . .

. . . . . .

Riassumendo

- Una figura si dice simmetriche se esiste una simmetria che la trasforma in se stessa.
- Una figura può avere più assi di simmetria.

- I poligoni regolari hanno tanti assi di simmetria quante sono i lati del poligono.
- La funzione `polreg(centro, vertice, numlati, **kargs)` può essere realizzata in questo modo:

```
def polreg(centro, vertice, numlati, **kargs):
    """Restituisce un poligono regolare
       dati il centro un vertice e il numero di lati."""
    # crea la circ. su cui sono disposti i vertici non visibile
    circ = ig.Circle(centro, vertice, visible=False)
    # calcola la lunghezza dell'arco tra due vertici consecutivi
    arco=2./numlati
    # crea la lista dei vertici che contiene l'argomento vertice
    vertici=[vertice]
    # aggiungi alla lista dei vertici tutti gli altri
    for cont in range(1, numlati):
        vertici.append(ig.PointOn(circ, cont*arco))
    # restituisci il poligono costruito con questi vertici
    return ig.Polygon(vertici, **kargs)
```

- La funzione `simmpoli(poligono, asse, **kargs)` può essere realizzata in questo modo:

```
def simmpoli(poligono, asse, **kargs):
    """Restituisce il simm. di un poligono rispetto a asse."""
    # crea una lista vuota che conterrà i vertici
    # del poligono simmetrico
    vertici_simm=[]
    # per ogni vertice del poligono originale, calcola il
    # simmetrico e aggiungilo alla lista dei vertici simmetrici
    for vertice in poligono.vertices:
        vertici_simm.append(simmpunto(vertice, asse))
    # restituisci il poligono costruito con questi vertici
    return ig.Polygon(vertici_simm, **kargs)
```

3.8.5 Equazioni di alcune simmetrie assiali

Avvia un nuovo programma e salvarlo con il nome: `simmas04_equazioni.py`. Scrivi la solita funzione `simmpunto`.

Nel programma principale crea:

- un piano interattivo;
- una retta x sovrapposta all'asse x;
- una retta y sovrapposta all'asse y;
- un punto `P` e visualizza le sue coordinate;
- il punto `P'` simmetrico di `P` rispetto all'asse x e visualizza le sue coordinate;
- il punto `P''` simmetrico di `P` rispetto all'asse y e visualizza le sue coordinate;

- muovi il punto P in varie posizioni e completa la seguente tabella:

punto	simmetrico rispetto asse x	simmetrico rispetto asse y
A (-4; 3)	A'(. . . . . ; . . . . .)	A'‘(. . . . . ; . . . . .)
B (1; -4)	B'(. . . . . ; . . . . .)	B'‘(. . . . . ; . . . . .)
C (. . ; . .)	C'(. . . . . ; . . . . .)	C'‘(. . . . . ; . . . . .)
P (x; y)	P'(. . . . . ; . . . . .)	P'‘(. . . . . ; . . . . .)

Nella simmetria rispetto all'asse delle `x`: l'ascissa del generico punto P' simmetrico di `P` è . . . . . all'ascissa di `P`; l'ordinata del generico punto `P'`, è . . . . . all'ordinata di `P`.

La simmetria rispetto all'asse x si può tradurre nel sistema di equazioni: $\sigma_{y=0} \begin{cases} x' = x \\ y' = -y \end{cases}$

Nella simmetria rispetto all'asse delle y:

. . . . . .

. . . . . .

La simmetria rispetto all'asse y si può tradurre nel sistema di equazioni: $\sigma_{x=0} \begin{cases} x' = \\ y' = \end{cases}$

Modifica il programma in modo che gli assi di simmetria coincidano con le bisettrici dei quadranti, muovi il punto P e completa la seguente tabella:

punto	simm. bis. I quadrante	simm. bis. II quadrante
A (-7; 3)	A'(. . . . . ; . . . . .)	A'‘(. . . . . ; . . . . .)
B (5; -2)	B'(. . . . . ; . . . . .)	B'‘(. . . . . ; . . . . .)
C (. . ; . .)	C'(. . . . . ; . . . . .)	C'‘(. . . . . ; . . . . .)
P (x; y)	P'(. . . . . ; . . . . .)	P'‘(. . . . . ; . . . . .)

Nella simmetria rispetto alla bisettrice del 1° e 3° quadrante: l'ascissa del generico punto `P'`, simmetrico di `P` è . . . . . `P`; l'ordinata del generico punto `P'`, è . . . . . .

La simmetria rispetto alla bisettrice del 1° e 3° quadrante si può tradurre nel sistema di equazioni: $\sigma_{y=x} \begin{cases} x' = \\ y' = \end{cases}$

Nella simmetria rispetto alla bisettrice del 2° e 4° quadrante:

. . . . . .

. . . . . .

La simmetria rispetto alla bisettrice del 2° e 4° quadrante si può tradurre nel sistema di equazioni: $\sigma_{y=-x} \begin{cases} x' = \\ y' = \end{cases}$

Modifica la funzione test in modo che gli assi di simmetria siano le rette di equazioni: $x = 3$ e $y = 4$. Muovi il punto P e completa la seguente tabella:

punto	simm. x = 3	simm. bis. y = 4
A (-6; 3)	A'(. . . . . ; . . . . .)	A''(. . . . . ; . . . . .)
B (4; -2)	B'(. . . . . ; . . . . .)	B''(. . . . . ; . . . . .)
C (. . ; . .)	C'(. . . . . ; . . . . .)	C''(. . . . . ; . . . . .)
P (x; y)	P'(. . . . . ; . . . . .)	P''(. . . . . ; . . . . .)

Nella simmetria rispetto alla retta x=3: l'ascissa del generico punto P', simmetrico di P è . . . . . P; l'ordinata del generico punto P', è . . . . . .

La simmetria rispetto alla retta x=3 si può tradurre nel sistema di equazioni: $\sigma_{x=3} \begin{cases} x' = \\ y' = \end{cases}$

In generale la simmetria rispetto alla retta x=k si può tradurre nel sistema di equazioni: $\sigma_{x=k} \begin{cases} x' = \\ y' = \end{cases}$

L'equazione di questa simmetria funziona anche se k=0? Cosa puoi osservare?

. . . . . .

. . . . . .

Nella simmetria rispetto alla retta y=4

. . . . . .

. . . . . .

La simmetria rispetto alla retta y=4 si può tradurre nel sistema di equazioni: $\sigma_{y=4} \begin{cases} x' = \\ y' = \end{cases}$

In generale la simmetria rispetto alla retta y=k si può tradurre nel sistema di equazioni: $\sigma_{y=k} \begin{cases} x' = \\ y' = \end{cases}$

L'equazione di questa simmetria funziona anche se k=0? Cosa puoi osservare?

. . . . . .

. . . . . .

Riassumendo

- Certe simmetrie assiali possono essere tradotte con un sistema di equazioni abbastanza semplice.
  - $\sigma_{y=0} \begin{cases} x' = x \\ y' = -y \end{cases}$
  - $\sigma_{x=0} \begin{cases} x' = -x \\ y' = y \end{cases}$
  - $\sigma_{y=x} \begin{cases} x' = y \\ y' = x \end{cases}$
  - $\sigma_{y=-x} \begin{cases} x' = -y \\ y' = -x \end{cases}$
  - $\sigma_{x=k} \begin{cases} x' = -x + 2k \\ y' = y \end{cases}$

- $\sigma_{y=k} \begin{cases} x' = x \\ y' = -y + 2k \end{cases}$

Prova tu

Sul quaderno completa le seguenti frasi.

1. Una simmetria assiale (s.a.) è
2. In una s.a. figure corrispondenti sono
3. In una s.a.:
   (a) sono punti uniti
   (b) sono rette unite
   (c) sono segmenti uniti
   (d) esiste una retta formata da tutti punti uniti, è:
4. I poligoni regolari hanno tanti assi di simmetria ...
5. Assi di simmetria...
   (a) il cerchio ha
   (b) il rettangolo ha
   (c) il rombo ha
   (d) il triangolo isoscele ha
   (e) il trapezio isoscele ha
   (f) Un parallelogramma che non sia rombo o rettangolo
6. Le equazioni della s.a.
   (a) rispetto all'asse x
   (b) rispetto all'asse y
   (c) rispetto alla bisettrice del 1° e 3° quadrante
   (d) rispetto alla bisettrice del 2° e 4° quadrante

3.9 Rotazione

In questo capitolo si affrontano i seguenti argomenti:

1. Cos'è una rotazione e quali sono le sue proprietà.
2. Cosa sono gli elementi uniti in una rotazione.
3. Cosa sono le rotazione di un poligono regolare.
4. Cosa dice l'algebra sulle rotazioni.

3.9.1 Definizione

Una rotazione rispetto a un centro `O` è una trasformazione che fa ruotare attorno a `O`, ogni punto del piano di uno stesso angolo,

Una rotazione è determinata dal centro e dall'angolo.

La funzione principale è quella che dato un *punto*, un *centro* e un *angolo* costruisce la rotazione del punto. Per cui:

```
p_1 = RuotaPunto(punto, centro, angolo)
```

Ovviamente `punto`, `centro` e `angolo` dovranno essere rispettivamente il punto che vogliamo trasformare, il centro di rotazione e l'angolo di rotazione creati precedentemente. Dopo la chiamata, `p_1` conterrà il riferimento al punto immagine di `p_0` nella rotazione.

La funzione RuotaPunto(punto, centro, ang) dovrà:

1. creare una semiretta invisibile passante per `centro` e `p_0`;
2. su questa semiretta riportare l'angolo;
3. intersecare questa semiretta con una circonferenza centrata in `centro` e passante per `p_0`;
4. dare come risultato questa intersezione.

Una possibile soluzione:

```
def ruotapunto(punto, centro, angolo, **kargs):
    """Restituisce la rotazione di punto dati centro e angolo."""
    lato_0 = ig.Ray(centro, punto, width=1)
    ang = ig.Angle(punto, centro, angolo)
    lato_1 = ang.side1(width=1)
    circ = ig.Circle(centro, punto, width=1)
    return ig.Intersection(circ, lato_1, 1, **kargs)
```

Avviato IDLE creiamo una nuova finestra (menu-File-New window) e la salviamo, in una nostra cartella, con il nome `rota01_proprieta.py`. Inizia questo programma con un'intestazione adeguata: alcuni commenti che contengano la *data*, il nostro *nome* e un *titolo*.

Il programma potrà assomigliare a questo:

```
# Rotazioni: proprietà

# lettura delle librerie
import pyig as ig

# funzioni
def ruotapunto(punto, centro, angolo, **kargs):
    """Restituisce la rotazione di punto dati centro e angolo."""
    lato_0 = ig.Ray(centro, punto, width=1)
    ang = ig.Angle(punto, centro, angolo)
    lato_1 = ang.side1(width=1)
    circ = ig.Circle(centro, punto, width=1)
```

```
    return ig.Intersection(circ, lato_1, 1, **kargs)

# programma principale
ip = ig.InteractivePlane()

# Creo l'asse di simmetria
centro = ig.Point(-3, -2, width=6, name='O')
angolo = ig.Angle(ig.Point(-5, 10, width=6),
                  ig.Point(-10, 10, width=6),
                  ig.Point(-6, 12, width=6), name='alfa')
angolo.side0(width=1)
angolo.side1(width=1)

# Punto A e il suo punto ruotato
a_0 = ig.Point(6, -1, width=6, name="A")
a_1 = ruotapunto(a_0, centro, angolo, width=6, name="A'")

# attivazione della finestra grafica
ip.mainloop()
```

Eseguiamo il programma, muoviamo i punti base, il punto `A'` deve corrispondere al punto `A` nella rotazione. Se tutto funziona siamo pronti per esplorare le caratteristiche delle rotazioni.

3.9.2 Proprietà

Cambia l'angolo di rotazione, cosa avviene quando è di 360°?

. . . . . .

Quando l'angolo di rotazione è un multiplo di 360° la rotazione diventa una particolare trasformazione: l'*identità*.

Costruisci ora un nuovo punto `B` e `B'`, il suo trasformato nella rotazione. Poi crea i segmenti `AB` e `A'B'` e visualizzane la misura. Puoi formulare la congettura: `A'B'` è congruente ad `AB`. Prova a dimostrarla.

. . . . . .

. . . . . .

. . . . . .

Costruisci un punto `P` vincolato al segmento `AB` e il suo simmetrico `P'`:

```
p = ig.ConstrainedPoint(ab, .3, width=6, color='olive drab', name="P")
p1 = simmpunto(p, asse, width=6, color='olive drab', name="P'")
```

Muovi il punto `P`, cosa osservi?

. . . . . .

Costruisci un nuovo punto `C` e `C'`, costruisci il poligono `ABC`, e il poligono `A'B'C'`. Cosa si può concludere circa i triangoli `ABC` e `A'B'C'`?

. . . . . .

Cosa puoi dire sull'orientamento dei vertici del triangolo ABC e del suo trasformato A′B′C′?

. . . . . .

Riassumendo

- La rotazione è una trasformazione geometrica che trasforma segmenti in segmenti congruenti, perciò è una *isometria*.
- La rotazione mantiene il verso dei poligoni.
- Se un punto appartiene ad un segmento, il suo ruotato appartiene al ruotato del segmento.
- Il programma completo:

```
# Rotazioni: proprietà

# lettura delle librerie
import pyig as ig

# funzioni
def ruotapunto(punto, centro, angolo, **kargs):
    """Restituisce la rotazione di punto dati centro e angolo."""
    lato_0 = ig.Ray(centro, punto, width=1)
    ang = ig.Angle(punto, centro, angolo)
    lato_1 = ang.side1(width=1)
    circ = ig.Circle(centro, punto, width=1)
    return ig.Intersection(circ, lato_1, 1, **kargs)

# programma principale
ip = ig.InteractivePlane()

# # Creo il centro e l'angolo di rotazione
centro = ig.Point(-3, -2, width=6, name='O')
angolo = ig.Angle(ig.Point(-5, 10, width=6),
                  ig.Point(-10, 10, width=6),
                  ig.Point(-6, 12, width=6), name='alfa')
angolo.side0(width=1)
angolo.side1(width=1)

# Punto A e A'
a_0 = ig.Point(6, -1, width=6, name="A")
a_1 = ruotapunto(a_0, centro, angolo, width=6, name="A'")

# Punto B e B'
b_0 = ig.Point(7, 3, width=6, name="B")
b_1 = ruotapunto(b_0, centro, angolo, width=6, name="B'")

# I segmenti AB, A'B' e le loro misure
ab =ig.Segment(a_0, b_0, width=6, color='violet')
a1b1 =ig.Segment(a_1, b_1, width=6, color='violet')
ig.VarText(-7, -7, "AB = {}", ab.length())
```

```
    ig.VarText(-7, -8, "A'B' = {}", a1b1.length())

    # P vincolato alla retta AB
    p_0 = ig.ConstrainedPoint(ab, .3, width=6,
                              color='olive drab', name="P")
    p_1 = ruotapunto(p_0, centro, angolo, width=6,
                     color='olive drab', name="P'")

    # Punto C, C', i triangoli ABC e A'B'C'
    c_0 = ig.Point(-1, 1, width=6, name="B")
    c_1 = ruotapunto(c_0, centro, angolo, width=6, name="C'")
    ig.Polygon((a_0, b_0, c_0), width=4, color='violet', intcolor='gold')
    ig.Polygon((a_1, b_1, c_1), width=4, color='violet', intcolor='gold')

    # attivazione della finestra grafica
    ip.mainloop()
```

3.9.3 Elementi uniti

Avvia un nuovo programma e salvarlo con il nome: `rota02_elementiuniti.py` e scrivi funzione `ruotapunto(punto, centro, angolo, **kargs)` che restituisce il corrispondente di un punto nella rotazione. Questa volta le linee di costruzione falle invisibili.

Quali sono gli elementi uniti di una rotazione?

. . . . . .

. . . . . .

Riassumendo

- In una trasformazione un elemento si dice unito se viene trasformato in se stesso.
- In una rotazione sono elementi uniti:
  - il punto . . . . . .
  - le circonferenze . . . . . .

3.9.4 Equazioni di alcune rotazioni

Avvia un nuovo programma e salvarlo con il nome: `rota03_equazioni.py`. Scrivi la solita funzione `ruotapunto(punto, centro, angolo, **kargs)`.

Nel programma principale crea:

- un piano interattivo;
- il centro di rotazione nell'origine degli assi;
- l'angolo di rotazione di 90°;
- un punto `P` e visualizza le sue coordinate;

- il punto `P'` e visualizza le sue coordinate;
- muovi il punto P in varie posizioni e completa la seguente tabella:

punto P	punto P'
P (-4; 3)	A'(. . . . . ; . . . . .)
P (1; -4)	B'(. . . . . ; . . . . .)
P (. . ; . .)	C'(. . . . . ; . . . . .)
P (x; y)	P'(. . . . . ; . . . . .)

Nella rotazione di 90° con centro nell'origine degli assi: l'ascissa del generico punto `P'` è . . . . . ; l'ordinata del generico punto `P'`, è . . . . . .

La rotazione di 90° con centro nell'origine si può tradurre nel sistema di equazioni: $\rho_{90} \begin{cases} x' = \\ y' = \end{cases}$

In modo analogo esplora le rotazioni di 180°, 270° e 360°.

. . . . . .

. . . . . .

. . . . . .

Riassumendo

- il programma per studiare le rotazioni di 90° può essere fatto così:

```
# Rotazioni: equazioni della rotazione

# lettura delle librerie
import pyig as ig

# funzioni
def ruotapunto(punto, centro, angolo, **kargs):
    """Restituisce la rotazione di punto dati centro e angolo."""
    lato_0 = ig.Ray(centro, punto, visible=False)
    ang = ig.Angle(punto, centro, angolo)
    lato_1 = ang.side1(visible=False)
    circ = ig.Circle(centro, punto, visible=False)
    return ig.Intersection(circ, lato_1, 1, **kargs)

# programma principale
ip = ig.InteractivePlane()

# Creo il centro e l'angolo di rotazione
centro = ig.Point(0, 0, width=6, name='O')
angolo = ig.Angle(ig.Point(-5, 10, visible=False),
                  ig.Point(-10, 10, visible=False),
                  ig.Point(-10, 12, visible=False), name='alfa')
angolo.side0(width=1)
angolo.side1(width=1)

# Punto P e P' e le loro coordinate
p_0 = ig.Point(6, -1, width=6, name="P")
```

```
    p_1 = ruotapunto(p_0, centro, angolo, width=6, name="P'")
    ig.VarText(-7, -11, "P = {}", p_0.coords())
    ig.VarText(-7, -12, "P' = {}", p_1.coords())

    # attivazione della finestra grafica
    ip.mainloop()
```

- Certe rotazioni possono essere tradotte con un sistema di equazioni abbastanza semplice.
  - $\rho_{90} \begin{cases} x' = \\ y' = \end{cases}$
  - $\rho_{180} \begin{cases} x' = \\ y' = \end{cases}$
  - $\rho_{270} \begin{cases} x' = \\ y' = \end{cases}$
  - $\rho_{360} \begin{cases} x' = \\ y' = \end{cases}$

Prova tu

Sul quaderno completa le seguenti frasi.

1. Una rotazione è
2. In una rotazione figure corrispondenti sono
3. In una rotazione:
   (a) sono punti uniti
   (b) sono circonferenze unite
4. Le equazioni di alcune rotazioni sono:

CAPITOLO 4

Il piano cartesiano con Python

4. Geometria analitica In questa sezione vengono proposti alcuni problemi di geometria analitica. Oltre a Python viene usata la libreria Pyplot di Pygraph. Pyplot mette a disposizione una finestra che contiene un piano cartesiano con la possibilità di disegnare gli assi e di far disegnare funzioni del tipo y=f(x), x=f(y) o polari: ro=f(th). Per l'installazione della libreria Pygraph, che contiene anche Pyplot, vedi l'introduzione.

4.1. Punti, segmenti, poligoni Come disegnare alcuni elementi geometrici nel piano cartesiano. La geometria analitica è uno strumento che permette di collegare due branche della matematica: l'algebra e la geometria. Permette quindi di affrontare problemi algebrici con metodi geometrici o viceversa problemi geometrici con strumenti algebrici. Lo strumento che permette di fare ciò è il riferimento cartesiano. La libreria pycart fornisce alcune la classe Plane dotata di alcuni metodi che permettono di disegnare punti, segmenti, poligoni, oltre agli assi cartesiani. Possiamo iniziare dall'ambiente IDLE per vedere come attivare le funzionalità di Pycart. Innanzitutto dobbiamo caricare l'oggetto Plane dalla libreria: >>> from pycart import Plane A questo punto, se tutto è andato bene, non deve essere successo niente. Se invece appaiono delle scritte rosse, vuol dire che qualcosa è andato storto, bisogna leggere il messaggio di errore e cercare di porre rimedio. In particolare bisogna essere sicuri di aver installato correttamente la libreria Pygraph. Poi dobbiamo creare un piano cartesiano su cui disegnare. Tecnicamente "istanziamo" un oggetto della classe Plane: >>> p=Plane() Questa volta, se tutto è andato bene, appare una finestra grafica vuota. La classe Plane ha diversi metodi e quindi possiamo dare all'oggetto p diversi comandi. Incominciamo con chiedere a p di tracciare gli assi cartesiani: >>> p.axes(True) Avendo passato l'argomento True, oltre agli assi, verrà disegnata anche una schiera di puntini. Ora possiamo assegnare a due variabili le coordinate di due punti: >>> p1=(2, 4) >>> p2=(-4, 3) Le variabili p1 e p2 sono associate ora a due oggetti tupla che in Python realizzano sequenze ordinate immutabili, nel nostro caso, delle coppie ordinate di numeri. Ora che abbiamo le coordinate dei punti che vogliamo tracciare non ci resta che dire all'oggetto p di disegnare i punti proprio lì: >>> p.drawpoint(p1) >>> p.drawpoint(p2) I punti vengono disegnati dove richiesto, ma può darsi che ci sembrino troppo piccoli e insignificanti. Possiamo decidere dimensioni e colore. Incominciamo ripulendo il piano cartesiano: >>> p.reset() Oltre a cancellare tutto quello che c'è nella finestra grafica, reset si premura di ridisegnare gli assi. Ora possiamo cambiare le dimensioni e il colore degli oggetti grafici che verranno disegnati: >>> p.setwidth(3) >>> p.setcolor("#5050a0") Il colore è nel formato: "#RRGGBB" dove RR, GG, BB sono tre numeri in formato esadecimale che rappresentano rispettivamente le componenti dei tre colori fondamentali: rosso, verde, blu. Ora possiamo disegnare i due punti: >>> p.drawpoint(p1) >>> p.drawpoint(p2) A questo punto se vogliamo disegnare un seg-

mento, magari giallino, possiamo dare al piano p i comandi: >>> p.setcolor("#a0a050") >>> p.drawsegment((-6, 8), (5, 7)) Da notare che il metodo drawsegment richiede le coordinate di due punti, cioè due coppie di numeri. Nel piano cartesiano un poligono può essere individuato dalle coordinate dei suoi vertici. L'istruzione: >>> poligono1=((-7, -5), (4, -7), (2, -2), (-3, -1)) assegna alla variabile poluigono1 una sequenza di coppie di numeri, per noi, una sequenza di coordinate. Per disegnare un poligono verdino possiamo usare il metodo drawpoly, dopo aver cambiato il colore della "penna": >>> p.setcolor("#50a050") >>> p.drawpoly(poligono1) drawpoly richiede come argomento una sequenza di coppie di numeri, in questo caso, la sequenza è contenuta nella variabile poligono1, ma avremmo anche potuto passarla direttamente scrivendo: >>> p.drawpoly(((-7, -5), (4, -7), (2, -2), (-3, -1))) In questo modo bisogna solo fare un po' più di attenzione al numero di parentesi. La coppia di parentesi più esterna contiene gli argomenti passati al metodo, la seconda contiene la lista di punti, le altre contengono le coordinate.

Riassumendo Il piano cartesiano presente nella libreria pygraph mette a disposizione vari metodi tra cui: setcolor("#rrggbb") dove rr, gg, bb, sono numeri esadecimali di 2 cifre, setwidth(<numero>) axes([True]) drawpoint(<coordinate>) drawsegment(<coordinate>, <coordinate>) drawpoly(<sequenza di coordinate>)

4.2. Trasformazioni Come trasformare poligoni. Nel piano cartesiano un poligono può essere individuato dai suoi vertici. Poiché possiamo rappresentare un punto con una coppia di numeri, una tupla di due elementi, un poligono può essere rappresentato da una sequenza di coppie di numeri. Iniziamo a scrivere la procedura test() che crea un piano cartesiano, un poligono e lo disegna. Ovviamente per prima cosa dobbiamo caricare dalla libreria la classe Plane:

from pycart import Plane

def test(): # crea un ogg. della classe Plot e disegna gli assi

piano=Plane("Trasformazioni", 600, 600, 15, 15) piano.axes(True)

 # predispone un pol. da cui partire per le trasformazioni poligono=((3, 4), (1, 5), (2, 3), (3, 3), (5, 1))

 # modifica spessore della penna e colore piano.setwidth(2) piano.setcolor("#a0a020")

 # disegna il poligono piano.drawpoly(poligono)

if __name__ == "__main__": test () L'ultima riga dice a Python che se questo file viene eseguito direttamente, non caricato come libreria, allora deve essere eseguita la funzione test(). Eseguendo il programma (<F5>) viene tracciato il poligono di vertici: (3, 4), (1, 5), (2, 3), (3, 3), (5, 1) con spessore della penna pari a 2 e di colore verde marcio. A questo punto affrontiamo le trasformazioni geometriche. La prima trasformazione che si studia di solito è la traslazione. Incominciamo da questa. Per traslare un intero poligono, basta traslare i suoi vertici, quindi iniziamo costruendo una funzione che ricevuti due argomenti, un punto e una traslazione restituisca il punto traslato. Tutte le prossime funzioni vanno scritte tra l'istruzione: "from pycart import Plane" e la linea: "def test():". Un commento che mette in evidenza la sezione relativa alla traslazione può essere utile. Se xp e yp sono le coordinate del punto e xt e yt sono le componenti della traslazione, le coordinate del punto traslato sono: xp+xt e yp+yt. In Python, per mettere i due elementi di una coppia in due variabili, è possibile usare gli indici: xp=punto[0] yp=punto[1] oppure usare una sintassi più diretta e, a mio avviso, più espressiva:

xp, yp = punto A questo punto possiamo realizzare la funzione traslapunto(punto, traslazione): ##### # Traslazione ###

def traslapunto(punto, traslazione): xp, yp = punto xt, yt = traslazione return xp+xt, yp+yt

Qui sono andato in crisi, è buona norma corredare tutte le funzioni di opportuni commenti, ma non sono riuscito a trovare qualche frase che potesse chiarire il significato più di quanto già faccia il codice. Il passo successivo è scrivere una funzione che, data una sequenza di punti e una traslazione, restituisca una sequenza che contenga i punti traslati. Anche in questo caso Python mette a disposizione due metodi. Con il primo si parte da una lista vuota "result" e , per ogni elemento della sequenza poligono, si aggiunge il suo traslato alla lista "result": def traslapoli(poligono, traslazione):

result=[] for vertice in poligono:

result.append(traslapunto(vertice, traslazione))

return result

Lo stesso effetto può essere ottenuto con un costrutto tipico di Python: la costruzione di liste: il risultato è la lista che ha per elementi i traslati in base a traslazione degli elementi di poligono: def traslapoli(poligono, traslazione):

return [traslapunto(vertice, traslazione) for vertice in poligono]

Molto sintetica, ma una volta che ci si abitua, anche molto espressiva. A questo punto non ci resta che aggiungere alla funzione test() le istruzioni per disegnare il poligono traslato. Dopo la riga "piano.drawpoly(poligono)" scriveremo: # Traslazione

piano.setcolor("#20a0a0") piano.drawpoly(traslapoli(poligono, (-10, 5)))

Ora abbiamo tutti gli strumenti per realizzare le altre trasformazioni che conosciamo.

Riassumendo Un punto può essere rappresentato da una coppia di numeri. Un poligono può essere rappresentato da una sequenza di coppie di numeri. Pycart mette a disposizione la classe Plain che, tra gli altri, ha i metodi: axes che disegna gli assi cartesiani, drawpoly che disegna un poligono. La trasformazione di un poligono può essere realizzata in due passi: la trasformazione di un punto, l'applicazione della trasformazione a tutti i punti del poligono.

4.3. Rette, parabole, cubiche... Come disegnare rette che attraversano il nostro piano cartesiano. L'attività svolta nei capitoli precedenti, usa i metodi della classe Plain definita nella libreria pycart. Se vogliamo disegnare il grafico di funzioni, dobbiamo usare i metodi forniti da un'altra classe definita nella libreria pyplot. Questa nuova classe è Plot che fornisce i metodi che permettono di tracciare funzioni di una variabile del tipo y=f(x) con il metodo xy("<funzione in x>") o x=f(y) usando il metodo yx("<funzione in y>"). Se vogliamo tracciare una retta possiamo usare uno dei due metodi precedenti. L'equazione esplicita della retta è: y=mx+q. quest'equazione permette di tracciare quasi tutte le rette del piano... ne restano escluse solo infinite, tutte quelle parallele all'asse y. Per tracciare la retta di equazione: devo 1. caricare la libreria pyplot, >>> from pyplot import * 2. predisporre il piano cartesiano, >>> p=Plot() 3. posso tracciare gli assi, che fanno sempre comodo, >>> p.axes(True) 4. inviare al piano il comando di rappresentare la retta: >>> p.xy("2/3*x+4") Viene tracciata una retta... peccato che non sia quella che ci aspettiamo, se il coefficiente angolare è 2/3, non dovrebbe essere parallela all'asse x! Il fatto è che Python, quando esegue una divisione tra numeri interi, calcola il quoziente troncando il numero senza i decimali. Il calcolo 2/3 in Python dà come risultato

0 per ottenere il numero razionale 2/3 devo riscrivere l'espressione in modo che o il numeratore o il denominatore sia un numero di tipo float: >>> p.xy("2./3*x+4") Per poter tracciare una retta parallela all'asse y devo usare il metodo che permette di tracciare funzioni del tipo x=f(y), quindi per disegnare la retta di equazione x=4 basta dare il comando: >>> p.yx("4") Per poter affrontare problemi più significativi, è utile una funzione che dati due numeri, il coefficiente angolare e il termine noto, produca la stringa che contiene la funzione della retta. La possiamo realizzare con il meccanismo della costruzione di stringhe usando i segnaposto "%s" come è spiegato con diversi esempi nei primi capitoli sulla programmazione in Python. >>> def retta(m, q):

return "%s*x+%s" % (m, q)

Facciamo una prova stampando la stringa ottenuta: >>> print retta(3./5, 6) 0.6*x+6 Possiamo ora usare questa funzione per tracciare una sequenza di rette con il coefficiente che va da 0 a 10. Innanzitutto ripuliamo il piano dalle rette precedenti: >>> p.reset() poi avviamo un ciclo che disegni tutte le rette richieste: >>> for m in range(11):

p.xy(retta(m, 4))

Se volessimo disegnare le rette con coefficiente angolare 1/2, 1/3, 1/4, ...1/10, la faccenda sarebbe un pochino più complicata per due motivi: Bisogna costringere Python a fare la divisione in virgola mobile e non troncando il risultato, Il primo valore di i è zero e quando nel ciclo si tenta di eseguire la divisione 1/i si ottiene un errore. La soluzione al primo problema si ottiene scrivendo 1./i invece che 1/i. Per risolvere il secondo problema, bisogna modificare il ciclo in modo che range restituisca gli interi a partire da 1 anziché da 0: range(1, 11): >>> for num in range(1, 11):

p.xy(retta(1./num, 4))

La funzione passata come argomento al metodo xy() ovviamente, può essere costruita a fantasia, basta che rispetti la sintassi di Python. Se il polinomio è di secondo grado otterremo una parabola con l'asse parallelo all'asse y: >>> p.xy("0.5*x*x-2./3*x+4") Anche qui, per i nostri esperimenti conviene costruire una funzione che dati i tre coefficienti del trinomio di secondo grado restituisca la stringa che lo rappresenta: >>> def parabola(a, b, c):

return "%s*x*x+%s*x+%s" % (a, b, c)

e possiamo provarla stampando il suo risultato: >>> print parabola(-.5, -1, 3) -0.5*x*x+-1*x+3 o chiedendo a pyplot di disegnarla per punti: >>> p.reset() >>> p.xy(parabola(-.5, -1, 3)) A questo punto possiamo costruirci gli strumenti per tracciare parabole cubiche o polinomi di 4°, 5°, 6°, ... grado osservando come cambia la forma del grafico al variare dei coefficienti.

Riassumendo Il metodo xy(<funzione>) della classe Pyplot permette di tracciare una qualunque funzione data nella forma y=f(x). La funzione passata al metodo come argomento deve essere una stringa. La stringa può essere scritta pari pari da tastiera oppure può essere costruita usando i metodi messi a disposizione da Python.

4.4. Rette ed estensione della classe Plot Come risolvere problemi di geometria analitica e come ampliare le capacità dei già potenti strumenti messi a disposizione da Pygraph. Finora abbiamo pasticciato scrivendo comandi nella finestra interattiva di IDLE, ora è giunto il momento di scrivere un programma. Scriviamo il programma che risolva un tipico problema di geometria analitica: calcolare l'equazione della retta passante per due punti e disegnarla. Prima di affrontare i problemi però decidiamo come rappresentare in Python una retta. La solita

equazione in due variabili in forma esplicita "y=mx+q" o implicita: "ax+by+c=0" non è comoda: è complicato estrarre le informazioni da una stringa scritta in questo modo. Come per rappresentare un punto usiamo una coppia di numeri, così possiamo usare una coppia di numeri (m, q) per rappresentare una retta. Dal capitolo precedente possiamo recuperare la funzione retta(m, q) che, dati i due coefficienti produca l'espressione eseguibile della funzione retta. Altra osservazione. Sarebbe comodo che gli oggetti della classe Plot avessero la capacità di disegnare rette, ma purtroppo non è così, hanno il metodo drawpoint, drawsegment, drawpoly, ma non un metodo drawline. Python permette di estendere facilmente le capacità di un oggetto. Possiamo costruire una classe che deriva tutte le caratteristiche della classe Plot e aggiunge ad essa il metodo drawline(retta). La nuova classe la chiameremo Mioplot. Una convenzione, accettata da tutti i pitonisti, prevede che i nomi delle classi inizino con la lettera maiuscola e noi ci adeguiamo volentieri, per rendere più comprensibile il nostro codice. Per dire a Python che vogliamo creare una nuova classe che estende le capacità di un'altra preesistente, possiamo usare la seguente sintassi: class Nuovaclasse(vecchiaclasse):

<nuovi metodi>

Non ci resta che incominciare a scrivere. Predisponiamo un nuovo file, salviamolo con il nome rette.py, scriviamo alcune righe di commento con le solite informazioni: autore, data, contenuto del file, ... Poiché la nostra nuova classe deriva dalla classe Plot contenuta nella libreria pyplot.py dobbiamo innanzitutto far caricare questa liberia con il comando: from pyplot import Plot Ora siamo pronti per creare la nuova classe: class Mioplot(Plot): La classe Mioplot ha tutte le caratteristiche e le capacità della classe Plot, ma noi vogliamo estenderle con dei nuovi metodi. Il primo serve per trasformare la coppia di numeri che rappresenta una retta in un'espressione lineare interpretabile da Python. I metodi di una classe hanno sempre, come primo parametro una variabile che conterrà il riferimento all'oggetto su cui il metodo deve lavorare. Per consuetudine a questo parametro viene dato il nome "self". Quindi il metodo retta deve far precedere i due parametri: coefficiente angolare e termine noto dal parametro self:

def retta(self, r):

"""Restituisce la stringa che contiene la funzione di una retta."""

return "%s*x+%s" % r

L'altro metodo che aggiungiamo a Plot serve per disegnare la retta rappresentata da una coppia di pun

def drawline(self, r):

"""Disegna una retta dati (coeff.angolare, termine noto)."""

self.xy(self.retta(r))

Bene non ci resta che scrivere la funzione che calcola la retta per due punti e poi provare tutto. Recuperiamo da qualche parte il modo per calcolare coefficiente angolare e termine noto di una retta che passi per due punti, qualcosa come: e : def retta2p(a, b):

"""Restituisce m e q della retta passante per a e b.""" xa, ya = a xb, yb = b m=float(yb-ya)/(xb-xa) q=ya-m*xa return m, q

Ed ora possiamo scrivere un test per le nostre funzioni: def test():

piano=Mioplot() # Crea un oggetto della classe Mioplot piano.axes(True) # chiede a piano di disegnare gli assi p0=(-4, -5) # Associa alle var. p0, p1, p2 tre p1=(4, -7) # coppie ordinate rappr. tre punti p2=(-4, 5) piano.setwidth(3) # aumenta le dim. degli ogg. disegnati piano.drawpoint(p0)#disegna i tre punti piano.drawpoint(p1) piano.drawpoint(p2) piano.setwidth(1) # riporta a 1 la dim. della "penna" piano.drawline(retta2p(p0, p1)) # disegna le tre rette piano.drawline(retta2p(p1, p2)) piano.drawline(retta2p(p2, p0))

Quando eseguiamo questo programma vogliamo che venga eseguita automaticamente la funzione test, aggiungiamo, infondo al file, la riga: if __name__ == "__main__": test () Bene dopo tutta questa attività di "coding" godiamoci il funzionamento del programma: premiamo <F5>, correggiamo gli eventuali errori di sintassi, se non succede niente controlliamo il testo del programma, se tutto va, vengono disegnati i tre punti e le tre rette. No, accidenti, non tre rette, ma solo due, al posto delle terza appare nella shell un messaggio di errore. (Se appare un messaggio di errore ma non appaiono i tre punti e le due rette nella finestra grafica, bisogna cercare di capire, trovare e correggere gli errori confrontando il programma eseguito con quello descritto sopra.) Leggiamo le ultime righe del messaggio:

File "/mnt/dati/daniele/06-07/scuola/materiali/informatica/pyplot/rette.py", line 49, in retta2p
m=float(yb-ya)/(xb-xa)

ZeroDivisionError: float division ci danno importanti informazioni sull'errore riscontrato. L'ultima riga ci informa che il nostro codice ha chiesto a Python di eseguire una divisione per zero, cosa non buona... La penultima riga riporta l'istruzione in cui si è verificato il fattaccio, la terzultima il numero di riga e la funzione in cui è scritto questo comando. Cerchiamo di capire cosa non va in questa funzione. Il problema non è di sintassi e non è un problema informatico ma geometrico. Veramente se si sceglie un'altra terna di punti magari il programma funziona senza errori. È un errore logico, gli errori logici sono i più difficili da individuare e da correggere, perché si manifestano solo in particolari condizioni non sempre facili da riprodurre. Nel nostro caso la retta passante per p0 e p2 è parallela all'asse y e non si può rappresentare con un'equazione esplicita, non è possibile calcolare il suo coefficiente angolare. Prima di eseguire la divisione conviene controllare che le ascisse dei due punti, xa e xb non siano uguali. Modifichiamo quindi la funzione retta2p. Dopo aver spacchettato le coordinate di a e b, controlliamo se xa è uguale a xb. In questo caso la coppia restituita conterrà come primo elemento, al posto del numero che rappresenta il coefficiente angolare, l'oggetto "None". Se invece xa e xb sono diversi, procede esattamente come prima: def retta2p(a, b):

"""Restituisce m e q della retta passante per a e b.""" xa, ya = a xb, yb = b if xa==xb:

return None, xa

else: m=float(yb-ya)/(xb-xa) q=ya-m*xa return m, q

Ora dobbiamo modificare anche i metodi della classe Mioplot in modo che tengano conto della possibili

def retta(self, r):

"""Restituisce la stringa che contiene la funzione di una retta."""

m, q = r if m==None:

```
        return "%s" % q
    else: return "%s*x+%s" % r
```

Il metodo che disegna la retta deve chiamare il metodo xy() nel caso di una funzione y=f(x) e il metodo y

```
def drawline(self, r):
    """Disegna una retta dati (coeff.angolare, termine noto)."""
    m, q = r if m==None:
        self.yx(self.retta(r))
    else: self.xy(self.retta(r))
```

Finalmente il problema di partenza è completamente risolto, non solo, abbiamo anche esteso le capacità dell'oggetto Plot presente nella libreria Pygraph. Forse si potrebbe scrivere all'autore della libreria per invitarlo ad aggiungere questa nuova funzionalità al modulo pyplot.py ;)

Riassumendo Con una coppia di numeri possiamo rappresentare quasi tutte le rette del piano cartesiano. Utilizzando anche un valore non numerico per il primo elemento della coppia, possono essere rappresentate tutte le rette. È possibile modificare le classi già definite nelle librerie estendendone le funzionalità. Ogni metodo di una classe ha come primo parametro un riferimento all'oggetto stesso normalmente chiamato "self". Abbiamo derivato dalla classe Plot la classe Mioplot che aggiunge due metodi: retta() e drawline(). Abbiamo costruito la funzione che date due coppie, le coordinate di due punti, restituisce una coppia, la retta passante per i due punti.

4.1 ...

...

...

Finalmente sono arrivato al termine di questo lavoro... "termine" si fa per dire. Molto altro si potrebbe scrivere e si potrebbero proporre molti altri esercizi. Anche la forma potrebbe essere molto migliorata: non sono certo esperto in impaginazione! Nonostante la cura che ho messo nel controllare contenuto e forma, molti errori mi saranno sfuggiti, ringrazio chiunque me li segnali. Ringrazio chi avrà avuto la forza di leggere questo testo o, magari, il coraggio di usarlo nella didattica, così come è, o solo in parte o, adattato alle proprie esigenze.

Mogo Kelen Te Sira Be!

Daniele Zambelli

CAPITOLO 5

La geometria della tartaruga

5.1 Procurarsi gli strumenti

Per prima cosa dobbiamo procurarci gli strumenti necessari:

- *Python*
- la libreria *pygraph*

5.1.1 Installare Python

Per prima cosa deve essere installato nel proprio sistema l'interprete del linguaggio di programmazione **Python**.

Per chi usa Windows, dal sito

www.python.org

ci si scarica l'ultima versione e la si installa.

Per gli altri, si chiede al proprio sistema di installare:

- *python*;
- *idle*;

5.1.2 Installare *pygraph*

Dal sito:

bitbucket.org/zambu/pygraph

si fa il dowload del pacchetto e lo si scompatta in una cartella.

Poi si copia la cartella *pygraph* e il file *pygraph.pth* nella cartella:

syte-packages del proprio Python.

Per Windows la cartella dovrebbe trovarsi nel percorso:

C:python3.3Libsyte-packages

5.1.3 I comandi di base

Idle

Il modo più semplice per scrivere un programma in Python è quello di usare l'interfaccia *Idle*.

Per cui dal menu-programmi-Python, si avvii *Idle*.

Idle ci permette di dare dei comandi e di vederne il risultato alla pressione del tasto <*Invio*>.

Ad esempio possiamo dare il comando:

```
fa qualcosa!
```

```
  File "<ipython-input-3-0ab3e73963c7>", line 1
    fa qualcosa!
               ^
SyntaxError: invalid syntax
```

In questo caso otteniamo un errore.

Un comando che dovrebbe capire è:

```
print(5)
```

```
5
```

questa volta è andato...

Al posto di 5 possiamo scrivere un'espressione complessa quanto vogliamo. Se è corretta verrà eseguita e verrà stampato il risultato.

Provate.

```
print(2**100)
```

```
1267650600228229401496703205376
```

Possiamo anche osservare che l'aritmetica dei numeri interi di Python prevede numeri limitati solo dalle capacità del computer. Pyton è in grado di calcolare anche 2^1000.

```
print(2**1000)
```

```
10715086071862673209484250490600018105614048117055336074437503883703510511249361
```

Oltre ai numeri interi Python opera anche con altri oggetti primitivi:

- interi;
- numeri con la virgola;
- stringhe;
- insiemi;

- tuple;
- liste;
- ...

Giusto per curiosità possiamo anche vedere che Python è in grado di fare operazioni piuttosto strane:

```
print('casa')
print('matta')
print('casa' + 'matta')
print('ciao')
print('ciao ' * 7)
```

```
casa
matta
casamatta
ciao
ciao ciao ciao ciao ciao ciao ciao
```

5.1.4 Altri problemi

1. Calcola la somma dei primi 20 numeri naturali.
2. Calcola il prodotto dei naturali dall'uno al venti.
3. Calcola l'area di un trapezio che ha: B = 15.3, b = 11.4, e h = 21.3
4. Confronta l'area ottenuta al punto precedente con le aree che ottieni diminuendo di un decimo o aumentando di un decimo le misure.
5. Fa calcolare a Python la soluzione di un problema di geometria.

5.2 Il primo programma

Ma basta divagazioni, presa confidenza con qualche comando Python passiamo a scrivere il nostro primo programma con la geometria della tartaruga.

Dal menu File sciegliamo New window.

Viene creata una finestra vuota dove scrivere il nostro programma.

Per prima cosa salviamo il programma con un nome **che termini con '.py'**.

Poi eseguiamolo premendo il tasto <*F5*>.

Se non compare nessuno strano messaggio vuol dire che non ci sono errori (e come potrebbero essercene dato che non abbiamo ancora scritto niente?).

Bene, incominciamo a riempire il nostro programma.

5.2.1 I commenti

Come prime istruzioni scriviamo qualcosa che non faccia assolutamente niente. In ogni buon programma devono esserci dei commenti e noi iniziamo il programma scrivendo, come commento, la data, il nostro nome e un titolo del nostro lavoro:

"Primo programma"

Per indicare a Python che la parte di riga che segue è un commento, si usa il carattere: #.

```
# 6 giugno 2014
# Daniele Zambelli
# Primo programma del corso Pas
```

Come prima eseguiamo il programma premendo <*f5*>. Se tutto fila liscio non dovrebbe succedere niente.

Se appaiono delle scritte rosse dobbiamo cercare di capire cosa Python tenta di dirci leggendo l'ultima linea.

5.2.2 Lettura delle librerie

Python è un programma di uso generale, ma noi vogliamo fargli prodirre la geometria della tartaruga. Per ottenere questo dobbiamo dire al programma di leggere la libreria che contiene gli oggetti negessari:

```
import pyturtle as pt
```

Eseguiamo il programma: <*F5*> e per prima cosa controlliamo che non siano apparsi strani messaggi rossi.

Ora cerchiamo di capire l'istruzione:

Viene letta la libreria e viene associata all'abbreviazione *pt*.

Il programma funziona, ma è piuttosto deludente: non appare ancora nulla!

Il nostro secondo comando produrrà un effetto visibile sullo schermo:

Vogliamo che venga creato un piano su cui far muovere delle tartarughe. A questo piano vogliamo dare un nome per poterci riferire a lui in seguito.

5.2.3 Creazione degli elementi di base

```
foglio = pt.TurtlePlane()
```

Eseguendo il programma (al solito con la pressione del tasto *F5*>, ma non lo dirò più), Appar una finestra vuota.

La finestra non risponde ai comandi del mouse, non si riesce neppure a chiudere.

Per renderla attiva dobbiamo comandarglielo:

```
foglio.mainloop()
```

Questo comando dovrà restare l'ultimo del nostro programma per cui ci portiamo tra questi due comandi e riprendiamo a scrivere...

Per poter far disegnare una figura alla tartaruga abbiamo bisogno di... una **tartaruga**!

Dovremo anche darle un nome, come abbiamo fatto per il piano. E allora creiamola. Il programma diventa:

```
import pyturtle as pt       # legge la libreria 'pyturtle' chiandola 'pt'
foglio = pt.TurtlePlane()   # crea un paese delle tartarughe
tina = pt.Turtle()          # crea una tartaruga

foglio.mainloop()           # rende attiva la finestra
```

5.2.4 I primo problema

Se tutto è andato bene, nessuno strano messaggio rosso, siamo pronti per concentrarci sulla geometria della tartaruga.

I comandi di base della geometria della tartruga sono:

- *forward(<num>)*;
- *back(<num>)*;
- *left(<num>)*;
- *right(<num>)*;
- *penup()*;
- *pendown()*;

Come primo problema facciamo disegnare a tina: un quadrato;

```
import pyturtle as pt       # legge la libreria 'pyturtle' chiandola 'pt'
foglio = pt.TurtlePlane()   # crea un paese delle tartarughe
tina = pt.Turtle()          # crea una tartaruga

tina.forward(50)        # va avanti di 50 passi
tina.left(90)           # gira a sinistra di 90 gradi
tina.forward(50)        # ...
tina.left(90)
tina.forward(50)
tina.left(90)
tina.forward(50)
tina.left(90)

foglio.mainloop()           # rende attiva la finestra
```

5.2.5 Altri problemi

1. Disegna le seguenti figure:
2. una bandierina;
3. una casetta;
4. una barchetta.
5. Inventa un semplice disegno e realizzalo.
6. Realizza alcuni semplici disegni nello stesso foglio.

5.3 Strutture di controllo

Nel primo programma che abbiamo costruito abbiamo utilizzato la struttura base di ogni programma: la **sequenza**.

Ogni programma può essere pensato come una sequenza di istruzioni, ma ci sono situazioni nelle quali la sequenza non è comoda.

In questo capitolo vedremo altre due strutture di controllo:

- l'iterazione,
- le funzioni,
- la selezione.

5.3.1 L'iterazione

Se voglio disegnare un quadrato devo ripetere 4 volte una coppia di comandi, ma se invece di un quadrato volessi far disegnare un "ottantagono"? La cosa non sarebbe più complessa, ma più noiosa sì!

Gli informatici, che sono ancora più pigri dei matematici, hanno inventato dei modi per evitare di ripetere istruzioni. Sono le struttre di *iterazione*.

In Python come negli altri linguaggi ci sono diverse struttre di controllo, ma noi vedremo l'istruzione *for*.

La sua sintassi è:

```
for <variabile> in range(<numero>):
    <istruzioni>
```

Un esempio forse può chiarire meglio il suo funzionamento:

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

La variabile *contatore* assume tutti i numeri compresi tra *0* e *5*, con *5* escluso, e ogni volta viene eseguito il blocco di istruzioni che segue il carattere *:*.

In questo esempio:

- *contatore* è il nome della variabile,
- *5* è il numero di ripetizioni,
- *print(contatore)* è il blocco di istruzioni che viene ripetuto.

Nota: in realtà le cose sono un po' più complesse, ma, per ora, possiamo accontentarci di questa spiegazione.

Vogliamo scrivere un programma che disegni un quadrato usando l'iterazione.

Per prima cosa dobbiamo scrivere i vari comandi di base...

Ma invece di riscriverli (la pigrizia!), usiamo un barbatrucco:

1. salviamo il programma precedente con un nuovo nome, ricordandoci che deve terminare con *.py*
2. svuotiamolo da dutte le istruzioni tranne quelle di base;
3. modifichiamo il titolo: "poligoni";
4. eseguiamolo per vedere se è tutto a posto;

Ora scriviamo un programma che disegni un quadrato usando l'iterazione di Python:

```
import pyturtle as pt       # legge la libreria 'pyturtle' chiandola 'pt'
foglio = pt.TurtlePlane()   # crea un paese delle tartarughe
tina = pt.Turtle()          # crea una tartaruga

for cont in range(4):       # ripete 4 volte il blocco seguente
    tina.forward(20)        # va avanti di 20 passi
    tina.left(90)           # gira a sinistra di 90 gradi

foglio.mainloop()           # rende attiva la finestra
```

5.3.2 Le funzioni

Se in un programma avessi bisogno di più quadrati, dovrei copiare in più punti le stesse tre righe. Questo è contrario all'etica della pigrizia, gli informatici hanno inventato un modo per associare un blocco di istruzioni ad un nome.

In Python la sintassi è:

```
def <none funzione>():
    <blocco di istruzioni>
```

In pratica possiamo associare alla parola "quadrato" le istruzioni per disegnare un quadrato:

```
def quadrato():
    """Fa disegnare un quadrato a tina."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(20)        # va avanti di 20 passi
        tina.left(90)           # gira a sinistra di 90 gradi
```

Si può osservare che la seconda riga della funzione è costituita da una stringa detta "doc-string". Questa stringa serve per documentare la funzione, non è obbligatoria, ma è fortemente raccomandata.

Il programma precedente può dunque diventare:

```
import pyturtle as pt       # legge la libreria 'pyturtle' chiandola 'pt'
foglio = pt.TurtlePlane()   # crea un paese delle tartarughe
tina = pt.Turtle()          # crea una tartaruga

def quadrato():
    """Fa disegnare un quadrato a tina."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(20)        # va avanti di 20 passi
        tina.left(90)           # gira a sinistra di 90 gradi

foglio.mainloop()           # rende attiva la finestra
```

Ma questo programma non disegna niente!

Cosa è successo?

Se lo osserviamo bene possiamo accorgerci che abbiamo definito come disegnare un quadrato, ma non gli abbiamo mai detto di disegnarlo.

dobbiamo aggiungere l'istruzione:

quadrato()

5.3.3 Struttura di un programma

Approfittiamo per ristrutturare il programma e dargli la forma che hanno di solito i programmi seri:

1. Intestazione;
2. lettura delle librerie;
3. definizioni;
4. programma principale.

```
# 6 giugno 2014
# Daniele Zambelli
# Primo programma del corso Pas

# lettura delle librerie
import pyturtle as pt       # legge la libreria 'pyturtle' chiandola 'pt'
```

```
# definizione delle funzioni
def quadrato():
    """Fa disegnare un quadrato a tina."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(20)        # va avanti di 20 passi
        tina.left(90)           # gira a sinistra di 90 gradi

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga
quadrato()                      # disegna un quadrato

foglio.mainloop()               # rende attiva la finestra
```

Possiamo richiamare una funzione anche dall'interno di un'altra funzione. Per esempio potremmo definire *bandierina* come un'asta seguita da un quadrato:

```
def bandierina():
    """Disegna una bandierina quadrata."""
    tina.forward(40)
    quadrato()
    tina.back(40)
```

Aggiungi questa funzione al programma (dove?) e fa disegnare una bandierina.

Poi usando l'iterazione fa disegnare una rosa di bandierine.

5.3.4 La selezione

Alle volte in un programma bisogna scegliere in base a una qualche condizione, se eseguire un blocco di codice o un altro.

La struttura per fare ciò è la selezione. La sua sintassi è:

```
if <condizione>:
    <istruzioni se vera>
[else:
    <istruzioni se falsa>]
```

Nota: la parte tra parentesi quadre è facoltativa.

Anche qui un esempio può chiarire:

```
numero = int(input('scrivi un numero: '))
if (numero % 2) == 0:
    print(numero, 'è pari')
else:
    print(numero, 'è dispari')
```

Questa struttura di controllo ci sarà utile, nella geometria della Tartaruga, quando costruiremo funzioni ricorsive.

5.3.5 Altri problemi

1. Senza cancellare i poligoni precedentemente disegnati, disegna anche altri poligoni regolari:
2. un triangolo;
3. un pentagono;
4. un esagono;
5. un ettagono;
6. ...
7. Disegna un quadrato formato da quattro quadrati.
8. Disegna il simbolo di pericolo radiazioni.
9. Ripeti più volte un percorso a casaccio.
10. Realizza bandierine triangolari o con altre forme.
11. Disegna una fila di bandierine.
12. Definisci una funzione che disegni un percorso a casaccio, poi richiamala all'interno di un ciclo.
13. Scrivi un programma che chiede un numero compreso tra 2 e 5 e disegna un poligono con il numero di lati digitato.

5.4 I parametri

Abbiamo visto un modo per scrivere una porzione di codice e riutilizzarla in punti diversi del programma. Ma nelle situazioni concrete sorge spesso l'esigenza di eseguire una porzione di codice con qualche piccola variazione.

Ad esempio noi potremmo avere bisogno di disegnare quadrati grandi e piccoli.

Scrivamo un programma che disegni due quadrati, uno grande, "quadratone" e uno piccolo, "quadratino".

```
# 6 giugno 2014
# Daniele Zambelli
# Programma che disegna quadrati di diverso lato

# lettura delle librerie
import pyturtle as pt       # legge la libreria 'pyturtle' chiandola 'pt'

# definizione delle funzioni
def quadratone():
    """Fa disegnare un quadratone a tina."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(100)       # va avanti di 100 passi
```

```
        tina.left(90)           # gira a sinistra di 90 gradi

def quadratino():
    """Fa disegnare un quadratino a tina."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(20)        # va avanti di 20 passi
        tina.left(90)           # gira a sinistra di 90 gradi

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga
quadratone()                    # disegna un quadratone
quadratino()                    # disegna un quadratino

foglio.mainloop()               # rende attiva la finestra
```

Funziona, ma le due funzioni sono quasi uguali e questo contraddice il principio di pigrizia.

Osservate le funzioni e evidenziate le differenze: a parte il nome l'unica differenza è il numero che indica la lunghezza del lato del quadrato.

Gli informatici hanno inventato un meccanismo per far disegnare quadrati grandi o piccoli alla stessa funzione.

Al posto del numero che cambia scriviamo un nome:

```
def quadrato():
    """Fa disegnare un quadratone a tina."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(lato)      # va avanti di lato passi
        tina.left(90)           # gira a sinistra di 90 gradi
```

Ma non basta, la funzione deve sapere che ha bisogno di conoscere la lunghezza del lato. Il meccanismo usato dagli informatici è quello aggiungere dei parametri alla funzione, la sintassi è:

```
def <nome funzione>(<elenco dei parametri>):
    <istruzioni>
```

Nel nostro caso, aggiungiamo alla funzione *quadrato* il parametro *lato*:

```
def quadrato(lato):
    """Fa disegnare un quadratone a tina."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(lato)      # va avanti di lato passi
        tina.left(90)           # gira a sinistra di 90 gradi
```

Se ora chiamiamo la funzione quadrato come nei programmi precedenti otteniamo un errore che ci dice più o meno che quadrato ha un parametro e che deve essere chiamato passandogli un argomento, cioè un valore da associare al parametro.

La chiamata di questa funzione dovrà essere qualcosa di simile a:

```
quadrato(37)
```

La lunghezza del lato del quadrato viene così decisa non quando viene definita la funzione, ma quando viene chiamata. La funzione *quadrato* disegnerà quindi un numero enorme di quadrati diversi a seconda del valore dell'argomento passato alla funzione.

Il programma precedente diventa quindi:

```
# 6 giugno 2014
# Daniele Zambelli
# Programma che disegna quadrati di diverso lato

# lettura delle librerie
import pyturtle as pt       # legge la libreria 'pyturtle' chiandola 'pt'

# definizione delle funzioni
# Invece di cancellare le seguenti due funzioni, ormai inutili,
# le ho commentate così da tenere traccia dell'evoluzione
# del programma.
#
#def quadratone():
#    """Fa disegnare un quadratone a tina."""
#    for cont in range(4):       # ripete 4 volte il blocco seguente
#        tina.forward(100)       # va avanti di 100 passi
#        tina.left(90)           # gira a sinistra di 90 gradi
#
#def quadratino():
#    """Fa disegnare un quadratino a tina."""
#    for cont in range(4):       # ripete 4 volte il blocco seguente
#        tina.forward(20)        # va avanti di 20 passi
#        tina.left(90)           # gira a sinistra di 90 gradi

def quadrato(lato):
    """Fa disegnare a tina un quadrato dato il lato."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(lato)      # va avanti di 20 passi
        tina.left(90)           # gira a sinistra di 90 gradi

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga
quadrato(100)                   # disegna un quadratone
quadrato(20)                    # disegna un quadratino

foglio.mainloop()               # rende attiva la finestra
```

5.4.1 Altri problemi

1. Disegna tre quadrati diversi in tre posizioni dello schermo.

2. Disegna 15 quadrati con lato da 10 a 150 uno dentro l'altro.
3. Disegna una spirale di quadrati.
4. Disegna una fila di quadrati. Ricordati di far tornare la tartaruga nella posizione iniziale.
5. Disegna una "coda" di quadrati, cioè una fila di quadrati non disposti in linea retta.
6. Scrivi le procedure che disegnano un triangolo e un quadrato con lato variabile. Confrontale. Scrivi la procedura che, dati *numlati* e *lato*, disegni un qualunque poligono regolare.
7. Usa la procedura precedente per realizzare una composizione grafica a fantasia.

5.5 Risolvere un problema

Risolvere problemi è una delle principali attività della nostra vita. Insegnare metodi efficaci per risolvere probremi dovrebbe essere uno dei principali obiettivi del nostro insegnamento.

5.5.1 Metodi di soluzione di problemi

Risolvere problemi è un'attività complessa quindi non esiste **il metodo** di soluzione dei problemi.

La geometria della tartaruga può aiutarci a imparare i metodi "top down" e "bottom up".

5.5.2 Top Down

È il metodo più razionale, va dalla soluzione del problema generale alla soluzione delle sue componenti. È adatto alla ricerca delle soluzioni di un problema complesso, si presta bene all'uso in gruppi dove il lavoro viene suddiviso in parti. Ma partiamo con un esempio.

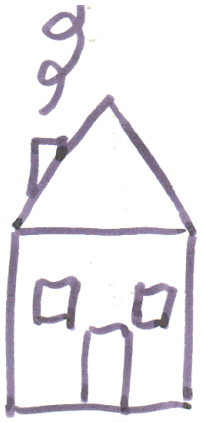
5.5.3 Il problema

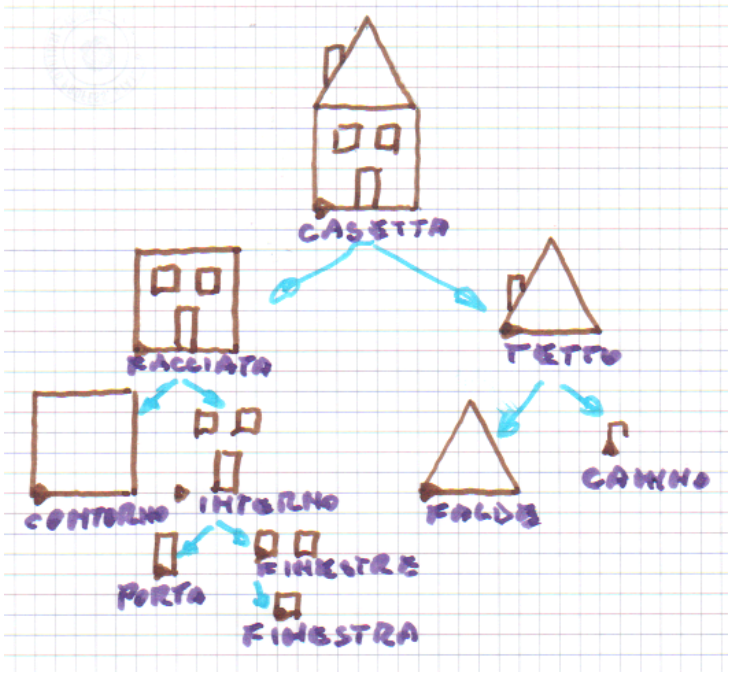
Voglio far disegnare a Tartaruga una casetta con porte finestre e camino. Qualcosa che assomigli a questo:

5.5.4 L'analisi

Prima di scrivere un programma che disegni la casetta, dobbiamo analizzarlo. L'analisi consiste nel suddividere il problema in parti più semplici e ripetere questa operazione in modo ricorsivo finché si arriva a parti elementari. Nel nostro caso una possibile analisi è:

Nell'analisi dobbiamo anche precisare: * Le dimensioni delle varie parti, nel nostro caso: un quadretto = 10 passi; * La posizione dove parte e dove arriva Tartaruga, normalmente queste due posizioni devono coincidere. * Un nome per ogni parte.




CASETTA
FACCIATA
TETTO
CAMINO
CONTORNO
INTERNO
FINESTRE
PORTA
FINESTRA

5.5.5 Grafo ad albero

Osservazione: l'analisi fatta in questo modo ha prodotto un oggetto che è studiato dalla matematica: un **grafo ad albero**.

Le varie parti della casetta costituiscono i *nodi*.

Le frecce costituiscono i *rami*.

Il nodo da cui parte tutto l'albero si chiama *radice*.

I nodi da cui non parte alcun ramo si chiamano *foglie*.

5.5.6 La soluzione

Fatta l'analisi possiamo iniziare a scrivere il nostro programma:

```
# 13 giugno 2014
# Daniele Zambelli
# Casetta

# lettura delle librerie
import pyturtle as pt           # legge la libreria 'pyturtle' chiandola 'pt'

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga

foglio.mainloop()               # rende attiva la finestra
```

Lo eseguiamo per assicurarci che non ci siano errori, ma già ci aspettiamo che non faccia un gran ché.

Passiamo a costruire la casetta, avendo deciso di seguire il metodo Top - Down, partiamo dal problema generale per scendere verso le varie componenti.

Quindi per prima cosa scriviamo la procedura casetta. Dall'analisi vediamo che è composta da due parti: * *facciata*, * *tetto*,

La funzione dovrà disegnare la facciata, poi spostarsi e disegnare il tetto, ma non è finita, poi ritornare dove era partita:

```
def casetta():
    """Disegna una casetta."""
    facciata()
    # sposta tina nella posizione adatta per disegnare il tetto
    tetto()
    # rimetti a posto tina
```

Guardando l'analisi non è difficile trovare quali istruzioni vanno scritte al posto dei commenti.

Il programma diventa:

```
# 13 giugno 2014
# Daniele Zambelli
# Casetta

# lettura delle librerie
import pyturtle as pt           # legge la libreria 'pyturtle' chiandola 'pt'

def casetta():
    """Disegna una casetta."""
    facciata()
    tina.left(90)
    tina.forward(50)
    tina.right(90)
    tetto()
    tina.left(90)
    tina.back(50)
    tina.right(90)

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga

foglio.mainloop()               # rende attiva la finestra
```

Lo eseguiamo... che delusione, non disegna proprio niente. Come mai?

Abbiamo definito casetta ma non abbiamo chiesto a Python di eseguire questa funzione, facciamolo:

```
# 13 giugno 2014
# Daniele Zambelli
# Casetta

# lettura delle librerie
import pyturtle as pt           # legge la libreria 'pyturtle' chiandola 'pt'

def casetta():
    """Disegna una casetta."""
    facciata()
    tina.left(90)
    tina.forward(50)
    tina.right(90)
    tetto()
    tina.left(90)
    tina.back(50)
    tina.right(90)

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga
casetta()                       # disegna una casetta
```

```
foglio.mainloop()                # rende attiva la finestra
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-12-9945567f031a> in <module>()
     20 foglio = pt.TurtlePlane()      # crea un paese delle tartarughe
     21 tina = pt.Turtle()             # crea una tartaruga
---> 22 casetta()                      # disegna una casetta
     23
     24 foglio.mainloop()              # rende attiva la finestra


<ipython-input-12-9945567f031a> in casetta()
      8 def casetta():
      9     """Disegna una casetta."""
---> 10     facciata()
     11     tina.left(90)
     12     tina.forward(50)


<ipython-input-6-1639e4a1c7a9> in facciata()
     40     """Disegna la facciata della casetta."""
     41     contorno()
---> 42     interno()
     43
     44 def contorno():


NameError: name 'interno' is not defined
```

Accidenti, otteniamo un errore!

L'ultima riga del messaggio di errore ci dà un'indicazione: *facciata* non è definita... Già, potevamo aspettarcelo, gli abbiamo insegnato a fare *casetta* ma non a fare *facciata*.

Rimettiamoci al lavoro e definiamo la nuova funzione. *facciata* risulta più semplice dato che le due parti da cui è composta hanno la stessa partenza, quindi non servono spostamenti intermedi. Aggiungiamo questa nuova funzione ed eseguiamo il programma:

```
# 13 giugno 2014
# Daniele Zambelli
# Casetta

# lettura delle librerie
import pyturtle as pt          # legge la libreria 'pyturtle' chiandola 'pt'

def casetta():
    """Disegna una casetta."""
    facciata()
    tina.left(90)
    tina.forward(50)
```

```
    tina.right(90)
    tetto()
    tina.left(90)
    tina.back(50)
    tina.right(90)

def facciata():
    """Disegna la facciata della casetta."""
    contorno()
    interno()

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga
casetta()                       # disegna una casetta

foglio.mainloop()               # rende attiva la finestra
```

Facendoci guidare dagli errori che man mano otteniamo, possiamo procedere a completare il programma.

Prima di procedere possiamo aggiungere 3 funzioni di supporto: *quadrato(lato)*, *triangolo(lato)*, *rettangolo(base, altezza)*:

Otteniamo ancora un errore, ma finalmente *tina* si è degnata di disegnare qualcosa.

L'interno della facciata è il più complesso sarà qualcosa di questo tipo:

```
def interno():
    """Dsegna l'interno della facciata."""
    # spostati in avanti
    # disegna la porta
    # spostati indietro e in alto senza tracciare segni
    # disegna le finestre
    # ritorna dove eri partita senza tracciare segni
```

Facendociguidare dagli errori, completiamo il disegno della casetta.

5.5.7 Il programma completo

```
# 13 giugno 2014
# Daniele Zambelli
# Casetta

# lettura delle librerie
import pyturtle as pt           # legge la libreria 'pyturtle' chiandola 'pt'

def quadrato(lato):
    """Fa disegnare a tina un quadrato dato il lato."""
    for cont in range(4):       # ripete 4 volte il blocco seguente
        tina.forward(lato)      # va avanti di lato passi
```

```
        tina.left(90)           # gira a sinistra di 90 gradi

def triangolo(lato):
    """Fa disegnare a tina un triangolo equilatero dato il lato."""
    for cont in range(3):       # ripete 3 volte il blocco seguente
        tina.forward(lato)      # va avanti di lato passi
        tina.left(120)          # gira a sinistra di 120 gradi

def rettangolo(base, altezza):
    """Fa disegnare a tina un rettangolo dato i lati."""
    for cont in range(2):
        tina.forward(base)
        tina.left(90)
        tina.forward(altezza)
        tina.left(90)

def casetta():
    """Disegna una casetta."""
    facciata()
    tina.left(90)
    tina.forward(50)
    tina.right(90)
    tetto()
    tina.left(90)
    tina.back(50)
    tina.right(90)

def facciata():
    """Disegna la facciata della casetta."""
    quadrato(5)                 # contorno
    interno()

def interno():
    """Dsegna l'interno della facciata."""
    tina.forward(20)            # spostati in avanti
    rettangolo(10, 20)          # disegna la porta
    tina.up()                   # spostati indietro e in alto
    tina.back(10)               # senza tracciare segni
    tina.left(90)
    tina.forward(30)
    tina.right(90)
    tina.down()
    finestre()                  # disegna le finestre
    tina.up()                   # ritorna dove eri partita
    tina.back(10)               # senza tracciare segni
    tina.right(90)
    tina.forward(30)
    tina.left(90)
    tina.down()

def finestre():
```

```
    """Disegna la porta della casetta."""
    quadrato(10)                # una finestra
    tina.up()                   # spostamento
    tina.forward(20)
    tina.down()
    quadrato(10)                # l'atra finestra
    tina.up()                   # ritorna dove eri partito
    tina.back(20)
    tina.down()

def tetto():
    """Disegna il tetto."""
    triangolo(50)               # falde del tetto
    tina.left(60)
    tina.forward(10)
    tina.left(30)
    camino()                    # camino
    tina.right(30)
    tina.back(10)
    tina.right(60)

def camino():
    """Disegna il camino."""
    tina.forward(20)            # avanti
    tina.right(90)
    tina.forward(7)
    tina.right(90)
    tina.forward(5)
    tina.back(5)                # ritorna per la stessa strada
    tina.left(90)
    tina.back(7)
    tina.left(90)
    tina.back(20)

# programma principale
foglio = pt.TurtlePlane()       # crea un paese delle tartarughe
tina = pt.Turtle()              # crea una tartaruga
casetta()                       # disegna una casetta

foglio.mainloop()               # rende attiva la finestra
```

5.5.8 Bottom up

Il metodo Bottom-up parte dalla stessa analisi, arriva allo stesso risultato (programma), ma segue un percorso diverso.

Invece che scrivere la funzione principale (radice) e farsi guidare dagli errori, si può partire dalla diverse componenti (foglie) realizzarle una alla volta ontrollando che rispondano alle specifiche dell'analisi e metterle insieme per ottenere il risultato desiderato.

5.5.9 Top down e problemi di matematica

Normalmente per risolvere problemi di matematica, nella scuola, si propone il meodo bottom up: si parte dai dati, si trova qualcosa di utile e via via cisi avvicina all'incognita. Questo metodo risulta di poco aiuto nella *ricerca* della soluzione.

Il metodo top down può fornire una guida preziosa nella soluzione dei problemi.

Prendiamo come esempio un problema di geometria solida:

Il volume di una piramide è $1000cm^3$ e la base è un rombo il cui perimetro è $52cm$ e una diagonale è di $24cm$. Calcola la misura dell'altezza.

Soluzione

$$altezza = \frac{3*volume}{sup.base}$$

$$sup.base = \frac{diag.1*diag.2}{2}$$

$$diag.1 = 2 * semid.1$$

$$semid.1 = \sqrt{lato^2 + semid.2^2}$$

$$semid.2 = \frac{diag.2}{2} = \frac{24cm}{2} = 12cm$$

$$lato = \frac{perimetro}{4} = \frac{52}{4} = 13cm$$

$$semid.1 = \sqrt{lato^2 + semid.2^2} = \sqrt{13^2 + 12^2} = 5cm$$

$$diag.1 = 2 * semid.1 = 2 * 5cm = 10cm$$

$$sup.base = \frac{diag.1*diag.2}{2} = \frac{10cm*24cm}{2} = 120cm^2$$

$$altezza = \frac{3*volume}{sup.base} = \frac{3*1000cm^3}{120cm^2} = 25cm$$

5.5.10 Altri problemi

1. Disegna una casetta con un'altra forma.
2. Disegna un albero.
3. Disegna un fiore.
4. Disegna una farfalla.
5. Riunisci alcuni elementi definiti sopra in un unico disegno.
6. Disegna una casetta a più piani.
7. Disegna una casetta con un numero variabile di piani: un grttacielo.

CAPITOLO 6

Foglio di calcolo

6. Foglio di calcolo Cosa c'è in questa sezione. In questa sezione sono viene introdotto brevemente l'uso del foglio di calcolo. Gli esempi e i termini usati si riferiscono al foglio di calcolo presente nel pacchetto OpenOffice.org, ma i problemi affrontati e le funzioni usate sono abbastanza semplici da permettere l'uso di un qualunque programma di foglio di calcolo.

6.1. Celle, colonne, righe... il foglio di calcolo Cos'è, e come usare le funzioni di base di un foglio di calcolo. Un Foglio di Calcolo è un'immensa tabella composta da alcune migliaia di righe e alcune centinaia di colonne che generano una grande quantità di celle nei loro incroci. L'elemento base di un Foglio di Calcolo, è dunque la cella. Ogni cella ha: un indirizzo, un contenuto e un formato: 1. indirizzo Come nella battaglia navale l'indirizzo di ogni cella è composto da una lettera seguita da un numero, ad es. B3 è la cella che si trova all'incrocio della seconda colonna con la terza riga. Poiché le lettere sono solo 26 e noi, a volte, abbiamo bisogno di più colonne, arrivati alla lettera "Z" proseguiamo con "AA", "AB"... e così via. Nella barra della formula, in alto a sinistra viene visualizzato l'indirizzo della cella in cui ci troviamo. Cliccando in diverse celle si può osservare l'indirizzo che cambia. 2. contenuto Ogni cella può avere un contenuto che è uno di questi 3 oggetti: Parole, una stringa qualunque. Numeri che possono rappresentare anche percentuali, ore o date. Formule, espressioni che iniziano con un uguale. Quando si termina di inserire una formula, nella cella viene mostrato il risultato del calcolo, mentre il testo della formula appare nella parte alta dello schermo nella barra di immissione. Gli operandi delle formule, possono essere numeri o indirizzi di celle. Quando viene modificato il contenuto di una cella, tutte le formule che contengono il suo indirizzo vengono ricalcolate. 3. formato Ogni cella ha diversi attributi che riguardano il suo formato o quello del suo contenuto. Ci sono decine di aspetti che possono essere modificati con il formato della cella: colore di sfondo; bordo; dimensioni; font, colore, dimensione dei caratteri; formato dei numeri; allineamento del contenuto; ... Possiamo applicare queste prime informazioni per realizzare un formulario di geometria che calcoli perimetri e aree di vari poligoni. Apriamo un nuovo foglio di calcolo. prima ancora di incominciare a riempirlo lo salviamo con nome: Menu-File-Salva Come. Conviene salvarle il documento in una nostra cartella e darle per nome "quadrilateri". Per salvare un file basta anche cliccare sull'icona con un dischetto, di solito terza da sinistra. L'obiettivo è avere un foglio nel quale inserire alcuni dati relativi ai quadrilateri notevoli e calcolare altre informazioni relative alla figura. Possiamo distinguere con un colore di sfondo le celle nelle quali inserire dati e con un altro colore quelle che conterranno i risultati. Dovremo adattare la larghezza delle colonne a seconda dello spazio occupato dal contenuto. Potrebbe anche essere utile graficamente separare i vari problemi riquadrando con un bordo le relative celle.

Cella Contenuto Formato A1 Formulario di geometria: i quadrilateri Dimensione e colore a fantasia A3 Problemi sul Quadrato Grassetto, colorato A5 Dato il lato trovo perimetro, area e diagonale del quadrato Grassetto, corsivo A6 Lato: allineamento a destra B6

Colore di sfondo: verde A7 Perimetro allineamento a destra B7 =B6*4 Colore di sfondo: azzurro A8 Area

B8 =B6^2 Colore di sfondo: azzurro A9 Diagonale

B9 =B6*sqrt(2) Colore di sfondo: azzurro A1:B9

Menu-Formato-Cella-Bordo: contorno

Prima di procedere con il formulario conviene provare inserendo nella cella B6 diversi valori numerici prima semplici per controllare che il foglio esegua calcoli corretti, poi più strani, con la virgola, molto grandi o molto piccoli e osservare i corrispondenti risultati. Una volta risolti eventuali problemi riscontrati possiamo passare ai problemi inversi o su altre figure.

Riassumendo Un foglio di calcolo è composto da un gran numero di "celle" organizzate in "righe" e "colonne" Ogni cella è caratterizzata da: un indirizzo, composto da una lettera o gruppo di lettere e un numero; un contenuto, che può essere: una frase, un numero, una formula; un formato

6.2. Formati e ordinamenti Come selezionare un blocco di celle, sommare i dati di un intero blocco, modificare la larghezza di una colonna, disegnare griglie, ordinare i dati. Spesso nei fogli di calcolo si devono inserire formule con molti operandi o molte formule che si assomigliano. I fogli di calcolo forniscono degli strumenti per scrivere formule applicate a grandi quantità di dati o scrivere, in modo efficiente e veloce, molte formule che si assomigliano. Come primo esempio partiamo dai dati relativi alla superficie dei continenti e alla loro popolazione, ma cerchiamo di ragionare pensando di avere a che fare con centinaia di righe di dati invece che con solo queste sette. Avviamo un nuovo foglio e salviamolo con il nome "continenti". Poi eseguiamo le seguenti istruzioni: Cella Contenuto Formato A1 Dati relativi alla popolazione e alla superficie dei continenti Dimensione e colore a fantasia A3:C10 Ricopiamo i dati della tabella riportata sopra

Non è difficile ricopiare la tabella precedente, si incontra qualche difficoltà nelle celle B3 e C3. La cella B3 contiene un carattere posto a indice, come ottenerlo? Innanzitutto si scrivono tutti i caratteri che vogliamo appaiano: "Area (km2)", poi con il mouse selezioniamo nella riga di immissione il solo carattere "2" e da Menu-Formato-Carattere-Posizione scegliamo "apice". Confermando con invio otteniamo il risultato desiderato. La cella C3 contiene una scritta troppo lunga che esce dai bordi della cella stessa, vorremmo che fosse spezzata su due righe. Poniamoci in C3 e modifichiamo il formato della cella: Menu-Formato-Celle-Allineamento-A capo automatico. Ora vogliamo che il contenuto di queste celle sia visualizzato in grassetto e sia centrato: dopo aver selezionato le celle, tra le icone che si trovano nella barra di formattazione troviamo i pulsanti giusti da cliccare per ottenere questi effetti. Possiamo ripetere queste operazioni per ognuna delle celle oppure... Gli informatici sono estremamente pigri (addirittura più dei matematici), poiché odiano ripetere le stesse operazioni e gli stessi gesti hanno inventato delle macchine bravissime a ripetere stupide operazioni. Invece che modificare per tre volte il formato di una cella è possibile selezionare le tre celle e aggiustarne il formato in un solo colpo. Per selezionare un gruppo di celle contiguo e rettangolare basta cliccare sulla cella in alto a sinistra e, tenendo premuto il tasto sinistro del mouse, trascinare il cursore fino alla cella in basso a destra. Quando si rilascia il tasto del mouse il colore delle celle selezionate apparirà

invertito. Ora vogliamo aggiungere una riga che contenga i totali della superficie e della popolazione: Cella Contenuto Formato B11 =somma(B4:B10) Grassetto C11 =somma(C4:C10) Grassetto

Se ora effettuiamo un doppio clic nella cella B11 ci viene evidenziata la formula e la zona di celle su cui lavora. Dato che la somma di un gruppo contiguo di celle è molto frequente, ci sono molti modi per immettere queste formule. Proviamo a vederle, poi, a seconda dei casi useremo quello più comodo. Cancelliamo il contenuto delle celle B11:C11. Ci riportiamo nella cella B11 e: iniziamo a scrivere la formula: "=somma(" selezioniamo con il mouse le celle B4:B10, chiudiamo la parentesi tonda e confermiamo con il tasto <Invio> Per la cella C11 proviamo ad usare un altro metodo. Una volta portati nella cella C11, clicchiamo l'icona della sommatoria che si trova in alto a sinistra della casella di inserimento se le scelte di Calc ci vanno bene, confermiamo la formula con il tasto <Invio>. I numeri, quando sono troppo lunghi, sono difficili da leggere e valutare, per facilitare questo compito si separano a gruppi di 3 con dei puntini i separatori delle migliaia (delle virgole per gli anglosassoni che usano invece il punto per separare la parte intera da quella decimale). Selezioniamo le celle da B4 a C11 e da Menu-Formato-Celle-Numeri sciegliamo il numero con il separatore delle migliaia e senza cifre decimali. A questo punto può succedere un effetto spiacevole, nella cella B11, dove prima c'era un numerone ora appaiono tre "cancelletti": "###". Cosa è successo? Modificando il formato del numero, ora la cella non è più abbastanza grande per contenerlo tutto. Poiché non è accettabile che un numero venga visualizzato solo in parte, quando non può essere contenuto in una cella, viene sostituito da un simbolo convenzionale: "###". Per vedere di nuovo il nostro numero possiamo seguire una delle seguenti strade: 1. togliere i puntini delle migliaia, 2. diminuire le dimensioni del carattere, 3. allargare la cella. La soluzione più adatta nel nostro caso è la terza. Clicchiamo con il tasto destro del mouse sull'intestazione della colonna da allargare e dal menu a tendina che appare scegliamo la voce: Larghezza colonna. Nel campo di inserimento al posto di 2, 62 scriviamo 3 e confermiamo. La colonna si sarà allargata un pochino e il numero verrà di nuovo visualizzato. A volte può essere utile avere i dati ordinati rispetto ad un certo criterio. Se i continenti fossero decine o centinaia, per trovare i dati relativi ad uno di questi sarebbe comodo averli scritti in ordine alfabetico. Ma questo ordine non mi servirebbe per trovare il più grande o il più popolato. Possiamo dire al foglio di calcolo di ordinare le righe che vogliamo in base al contenuto di una colonna. Se vogliamo ottenere i continenti in ordine alfabetico selezioniamo il blocco di celle da A4 a C10 e attraverso il Menu-Dati-Ordina scegliamo come primo criterio la colonna A. Confermando otteniamo le righe ordinate, in ordine alfabetico dall'Africa all'Oceania. Se vogliamo i continenti ordinati dal più popolato al meno popolato, sempre dopo aver selezionato tutte le celle che contengono i dati da ordinare, scegliamo dal Menu-Dati-Ordina come primo criterio la colonna C e come ordine quello discendente. In un batter d'occhio ritroveremo i nostri dati ordinati per popolazione.

Riassumendo Con il mouse è possibile selezionare un blocco di celle. È possibile assegnare un formato a tutte le celle di un blocco. È possibile calcolare la soma dei numeri contenuti in blocchi di celle. È possibile disegnare i contorni delle celle. Si può ordinare un blocco di celle in base a diversi criteri. Spesso ci sono molti modi diversi per eseguire la stessa operazione. È importante saper usarne uno, poi gli altri si imparano con il tempo e con l'uso.

6.3. Copiare in modo intelligente Come ricopiare formule usando indirizzi relativi e assoluti. Riprendiamo i dati già usati nel capitolo precedente, con delle semplici formule possiamo ottenere delle informazioni nuove. Possiamo, ad esempio, far calcolare la densità di popolazione per mezzo della formula popolazione/superficie. Cella Contenuto For-

mato D3 Densità ab/km2 Centrato, grassetto D3 (Selezionare nella riga di input il solo 2) Formato-carattere-posizione-apice D3

Formato cella-allineamento-acapo automatico D4 =C4*1000000/B4 Formato-celle-numeri-zero decimali D5 =C5*1000000/B5 Formato-celle-numeri- zero decimali D6 ... ...

Dato che i continenti sono solo 7 non è un grande problema scrivere le 7 formule diverse una sotto l'altra, ma in un foglio di calcolo non è infrequente dover scrivere centinaia o migliaia di formule sili a queste! Chi ha progettato il foglio di calcolo ha previsto degli strumenti che permettono di ricopiare velocemente delle formule. Ponendoci nella cella D4, appare nell'angolo in basso a destra, della cella stessa, un quadratino nero; con il mouse trasciniamo questo quadratino verso il basso fino a coprire tutte le celle in cui vogliamo ricopiare la formula. Non solo il programma ha ricopiato la formula ma ha anche aggiustato gli indici, proprio come ci serviva. Da notare che quando viene ricopiata una formula vengono anche ricopiati i formati della celle in cui la formula è stata scritta. Un'altra informazione interessante che possiamo ricavare dai pochi dati in nostro possesso è la percentuale rappresentata dalla superficie di un continente rispetto alla superficie totale delle terre emerse. La percentuale non è altro che un rapporto, il quoziente tra la superficie di un continente e il totale. Procediamo con il lavoro: Cella Contenuto Formato E3 Perc. Sup. Centrato, grassetto E4 =B4/B11

Il risultato di questo calcolo è 0,29, non è certo la percentuale cercata, se lavoriamo sulla carta, per trasformare questo numero nella percentuale basta moltiplicarlo per 100. Nei fogli di calcolo basta indicare nel formato della cella che quel numero deve essere inteso come una percentuale: Cella Contenuto Formato E4 =B4/B11 Formato-celle-numeri-percentuale E5 =B5/B11 Formato-celle-numeri-percentuale ... ...

Anche qui possiamo sfruttare il meccanismo di far ricopiare la formula verso il basso. Dopo esserci posizionati nella cella E4, prendiamo il quadratino e trasciniamolo verso il basso in modo da coprire le celle di tutti i continenti. Questa volta l'effetto non è quello desiderato: otteniamo una serie di errori! Come mai? Osserviamo una delle celle in cui è comparso l'errore, la cella E5 contiene la formula =B5/B12. Per capire meglio la formula selezioniamo la cella con un doppio clic. Vengono evidenziate in rosso e blu le celle che sono utilizzate nella formula stessa. Ora, B5 va bene, ma B12 doveva essere B11! Nella cella B12 non c'è niente e il foglio di calcolo segnala giustamente un errore di divisione per 0. Noi vogliamo che, nel ricopiare le formule, l'indice numerico di B4 venga modificato ma quello di B11 rimanga costante. Nei termini dei fogli di calcolo si dice che B4 deve essere un indirizzo relativo, B11 un indirizzo assoluto. Per essere pignoli a noi non occorre che tutto B11 ria assoluto, siccome vogliamo ricopiare la formula verso il basso ci basta che sia assoluta la parte numerica dell'indirizzo: l'11. Per comunicare questi desideri al foglio di calcolo si mette davanti al riferimento che vogliamo sia assoluto il carattere dollaro: "$". Questo fa si che il programma quando ricopia le formule non ne modifichi il riferimento. Aggiustiamo le nostre formule: Cella Contenuto Formato E4 =B4/B$11 Formato-celle-numeri-percentuale

Ora ricopiare la cella verso il basso produce l'effetto desiderato! Nella cella E5 ci sarà la formula =B5/B$11, nella cella E6 la formula =B6/B$11, e così via. L'elaborazione numerica dei nostri dati è completa, disegniamo una griglia anche attorno alle nuove celle che abbiamo riempito ottenendo così un lavoro presentabile.

Riassumendo Si possono "ricopiare" formule trascinando il quadratino che appare in basso a destra di una cella selezionata. Quando ricopiamo una formula verticalmente gli indici relativi alla riga, i numeri, vengono modificati. Quando ricopiamo una formula orizzontalmente gli

indici relativi alla colonna, le lettere, vengono modificati. Se vogliamo che, nel ricopiare una formula, un indice non venga modificato, basta che lo facciamo precedere dal carattere: "$".

6.4. Diagrammi Come rappresentare graficamente i dati. Spesso un grafico dà una più immediata comprensione di un fenomeno rispetto ad una lista di numeri. I fogli di calcolo permettono di disegnare grafici di diversa forma. Riprendendo il foglio dei continenti vogliamo aggiungere due grafici per rappresentare la superficie e la popolazione. Apriamo il foglio su cui abbiamo lavorato finora selezioniamo le celle che contengono i dati che vogliamo rappresentare. Iniziamo costruendo un grafico a torta che riporti la superficie dei diversi continenti. 1. Selezioniamo le celle comprese tra A4 e B10. 2. Da menu scegliamo Inserisci-Diagramma, viene così aperta una finestra di dialogo che ci guida nella definizione del diagramma. 3. Controlliamo che sia selezionata la casella "Prima colonna come didascalia" e premiamo "Avanti". 4. Nella seconda pagina di questo dialogo selezioniamo: "Rappresenta oggetti nell'anteprima". e scegliamo il grafico a torta e serie di dati in Colonne. 5. Nella terza pagina, scegliamo "normale". 6. Nell'ultima pagina scriviamo il titolo (ad es. "Superficie") e confermiamo cliccando sul bottone "Crea". A questo punto viene creato un diagramma. Un clic sul diagramma lo selezione e fa apparire le maniglie di dimensionamento che permettono di modificarne le dimensioni. Quando è selezionato possiamo anche spostarlo dove vogliamo che appaia nella nostra pagina. Posizioniamolo subito sotto ai dati. Questo è un buon momento per salvare il lavoro fatto. È possibile modificare i dati rappresentati nel diagramma cliccando con il tasto sinistro sul diagramma stesso e scegliendo, dal menu contestuale, la voce Modifica area dati. Se vgliamo che il diagramma sia riquadrato da un bordo, dopo aver dato un doppio clic sul diagramma, scegliamo dal menu contestuale la voce Area del diagramma. Se vogliamo modificare più profondamente il diagramma appena creato possiamo effettuare un doppio clic sul diagramma stesso. Il menu principale del foglio di calcolo cambia e cambiano anche i menu contestuali (quelli legati al tasto sinistro) a seconda di cosa viene puntato dal mouse. Dal menu Inserisci scegliamo Legenda e togliamo il segno di spunta su Visualizza. La Legenda scompare, ma adesso il diagramma è di difficile interpretazione, operiamo dunque un'altra modifica: sempre dal menu Inserisci scegliamo Etichette e chiediamo che ci vengano mostrati i valori come percentuale e anche le etichette di testo. Etichette troppo lunghe sbilanciano la rappresentazione conviene abbreviarle, nella tabella modifichiamo "America settentrionale in "America sett." e "America meridionale" in "America mer.". Alla conferma di questi cambiamenti anche le etichette nel diagramma saranno automaticamente modificate. Ora se il diagramma risulta troppo piccolo e non riempie bene lo spazio a sua disposizione possiamo cliccare vicino alla "torta" e allargarlo agendo sulle maniglie verdi che appaiono. Questo è un buon momento per salvare il lavoro fatto. Ora se vogliamo un diagramma che contenga i dati relativi al numero di abitanti dobbiamo selezionare i nomi dei continenti e i valori della popolazione. Purtroppo questi valori non sono contigui, per farlo dobbiamo usare un trucco: selezioniamo con il mouse le celle dalla A4 alla A10 e poi selezioniamo le celle dalla C4 alla C10 tenendo premuto contemporaneamente il tasto <Ctrl>. Il tasto <Ctrl> permette di effettuare selezioni multiple su blocchi rettangolari non contigui. Dopo aver selezionato l'area contenente i dati, dal menu-Insersci scegliamo la voce Diagramma. Questa volta invece che un diagramma a torta vogliamo un istogramma. Come prima assicuriamoci che sia selezionata la voce "Prima colonna come didascalia", nella pagina seguente selezioniamo "Rappresenta oggetti nell'anteprima". Possiamo così accorgerci che la legenda, in questo caso non ha senso. Nell'ultima pagina Scriviamo il titolo del diagramma: "Popolazione" e deselezioniamo la voce "Legenda". A questo punto possiamo creare il diagramma. Posizioniamolo in fianco al precedente. Vogliamo ora disegnargli un riquadro attorno: doppio clic, menu-Formato-Area del Diagramma, ... Vogliamo anche che le etichette

dell'asse x vengano scritte in verticale in modo da non essere spezzate: menu-Formato-Assi-AsseX e lì modifichiamo le etichette mettendo la rotazione a 90° selezionando "Sovrapponi" e deselezionando "A capo". Questo è un buon momento per salvare il lavoro fatto. Ora i diagrammi sono come li volevamo, Prima di considerare finito il lavoro dobbiamo però controllare di poterlo stampare in un'unica pagina. Clicchiamo fuori dai diagrammi, in una cella qualunque, poi da Menu Visualizza scegliamo "Interruzioni di pagina". Una linea blu delimiterà i contorni delle varie pagine, modifichiamo le dimensioni dei diagrammi o spostiamoli in modo da farli rientrare tutti in un'unica pagina, assieme ai dati. Se la scala della visualizzazione si è troppo ridotta possiamo cliccare con il destro sulla percentuale presente nella barra di stato (in basso) e scegliere il valore "100%". Possiamo anche agire sul formato della pagina: Menu-Formato-Pagina dove possiamo agire sull'orientamento della pagina (verticale o orizzontale), sui margini (possiamo ridurli per lasciare più posto ai contenuti) sull'intestazione o sul piè di pagina: togliamo l'intestazione e modifichiamo il piè di pagina scrivendo a sinistra la data e a destra il nostro nome. Un'occhiata al lavoro svolto con l'anteprima di stampa può rassicurarci che è tutto disposto per bene nella pagina. Se siamo soddisfatti possiamo considerare finito il lavoro, altrimenti chiudiamo l'anteprima e modifichiamo gli aspetti che non ci piacciono. Salviamo ancora una volta il lavoro ed eventualmente stampiamolo.

Riassumendo Il modo più semplice per realizzare un diagramma è quello di selezionare i dati che vogliamo rappresentare e poi scegliere Menu-Inserisci-Diagramma. Nel dialogo di costruzione di un diagramma possiamo scegliere diverse caratteristiche: etichette, tipo e sottotipo, assi, legenda, titoli, ... Una volta costruito un diagramma è possibile modificarlo usando il menu che appare dopo aver effettuato un doppio clic sul diagramma stesso. È Importante salvare spesso il proprio lavoro. La vista con interruzioni di pagina permette di impaginare in modo efficace il nostro lavoro. Il menu-Formato-Pagina permette di intervenire sull'orientamento, le dimensioni, i margini, le intestazioni, i piè di pagina, ...

CAPITOLO 7

...

...

...

Finalmente sono arrivato al termine di questo lavoro... “termine” si fa per dire. Molto altro si potrebbe scrivere e si potrebbero proporre molti altri esercizi. Anche la forma potrebbe essere molto migliorata: non sono certo esperto in impaginazione! Nonostante la cura che ho messo nel controllare contenuto e forma, molti errori mi saranno sfuggiti, ringrazio chiunque me li segnali. Ringrazio chi avrà avuto la forza di leggere questo testo o, magari, il coraggio di usarlo nella didattica, così come è, o solo in parte o, adattato alle proprie esigenze.

Mogo Kelen Te Sira Be!

Daniele Zambelli

CAPITOLO 8

Indices and tables

- genindex
- modindex
- search