Esercitazione: Turtle graphics

Questa esercitazione illustra il modulo grafico `turtle` che vi permette di creare immagini utilizzando *turtle graphics*. Si tratta di un modulo già compreso nella maggior parte delle installazioni di Python.

Il modulo turtle

Per controllare se il modulo `turtle` è installato, aprite l'interprete Python e scrivete:

```
>>> import turtle
>>> bob = turtle.Turtle()
```

Eseguendo questo codice, dovrebbe comparire una nuova finestra con un cursore a forma di freccetta che rappresenta un'ideale tartaruga. Ora chiudete pure la finestra.

Create un file di nome `griglia.py` e scriveteci il seguente codice:

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

Il modulo `turtle` (con la 't' minuscola) contiene un comando di nome `Turtle` (con la 'T' maiuscola) che crea un oggetto Turtle (una "tartaruga"); questo oggetto viene assegnato a una variabile di nome `bob`. Stampando `bob` viene visualizzato qualcosa di questo genere:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

Ciò significa che `bob` fa riferimento ad un oggetto `Turtle`, come definito nel modulo `turtle`.

`mainloop` dice alla finestra di attendere che l'utente faccia qualcosa, sebbene in questo caso non ci sia molto da fare, se non chiudere la finestra.

Una volta creata una tartaruga, potete chiamare uno dei suoi **metodi** per spostarla in giro per la finestra. Un metodo è simile ai comandi come `print`, `type` o `int` che abbiamo visto nella prima lezione, ma usa una sintassi leggermente diversa. Ad esempio, per spostare la tartaruga in avanti:

```
bob.fd(100)
```

Il metodo, `fd`, è associato all'oggetto Turtle che abbiamo chiamato `bob`. Chiamare un metodo è come effettuare una richiesta: in questo caso state chiedendo a `bob` di muoversi in avanti [fd sta per *forward*, NdT]. L'argomento di `fd` è una distanza espressa in pixel, per cui l'effettivo spostamento dipenderà dalle caratteristiche del vostro schermo.

Altri metodi che potete chiamare su una tartaruga sono: `bk` per muoversi indietro (*backward*) e `lt` e `rt` per girare a sinistra (*left*) e a destra (*right*). Per questi ultimi due, l'argomento è un angolo espresso in gradi.

Inoltre, ogni tartaruga regge una penna, che può essere appoggiata o sollevata; se la penna è appoggiata, la tartaruga lascia un segno dove passa. I metodi `pu` e `pd` stanno per "penna su (*up*)" e "penna giù (*down*)".

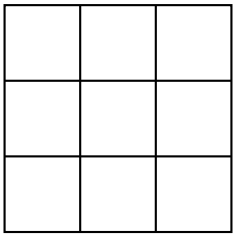
Per disegnare un angolo retto, aggiungete queste righe al programma (dopo aver creato `bob` e prima di chiamare `mainloop`):

Questa esercitazione è un adattamento del Capitolo 4 di "Pensare in Python" di A. Downey, traduzione di A. Zanella. `https://github.com/AllenDowney/ThinkPythonItalian`

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

Avviando il programma, dovreste vedere `bob` muoversi verso destra e poi in alto, lasciandosi dietro due segmenti.

Ora provate a modificare il programma in modo da disegnare una griglia 3×3 come quella mostrata qui sotto. Ogni cella della griglia è larga 100 pixel e alta 100 pixel.



Funzioni

Nell'ambito della programmazione, una **funzione** è una serie di istruzioni che esegue un calcolo, alla quale viene assegnato un nome. Per definire una funzione, dovete specificarne il nome e scrivere la serie di istruzioni. In un secondo tempo, potete "chiamare" la funzione mediante il nome che le avete assegnato.

Chiamate di funzione

Abbiamo già visto un esempio di una **chiamata di funzione**:

```
>>> type(42)
<class 'int'>
```

Il nome di questa funzione è `type`. L'espressione tra parentesi è chiamata **argomento** della funzione, e il risultato che produce è il tipo di valore dell'argomento che abbiamo inserito.

Si usa dire che una funzione "prende" o "riceve" un argomento e, una volta eseguita l'elaborazione, "ritorna" o "restituisce" un risultato. Il risultato è detto **valore di ritorno**.

Python contiene una raccolta di funzioni per convertire i valori da un tipo a un altro. La funzione `int` prende un dato valore e lo converte, se possibile, in un numero intero. Se la conversione è impossibile, Python comunica che si è verificato un errore:

```
>>> int('32')
32
>>> int('Ciao')
ValueError: invalid literal for int(): Ciao
```

`int` può anche convertire valori in virgola mobile in interi, ma non arrotonda bensì tronca la parte decimale.

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

La funzione `float` converte interi e stringhe in numeri a virgola mobile:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Infine, `str` converte l'argomento in una stringa:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Note provenienti dal Capitolo 3 di "Pensare in Python" di A. Downey, traduzione di A. Zanella. https://github.com/AllenDowney/ThinkPythonItalian

Funzioni matematiche

Python è provvisto di un modulo matematico che comprende buona parte delle funzioni matematiche d'uso frequente. Un **modulo** è un file che contiene una raccolta di funzioni correlate.

Prima di poter usare le funzioni contenute in un modulo, lo dobbiamo importare con un'**istruzione di importazione**:

```
>>> import math
```

Questa istruzione crea un **oggetto modulo** chiamato math. Se visualizzate l'oggetto modulo, ottenete alcune informazioni a riguardo:

```
>>> math
<module 'math' (built-in)>
```

L'oggetto modulo contiene le funzioni e le variabili definite all'interno del modulo stesso. Per accedere a una funzione del modulo, dovete specificare, nell'ordine, il nome del modulo e il nome della funzione, separati da un punto. Questo formato è chiamato **notazione a punto** o *dot notation*.

```
>>> rapporto = potenza_segnale / potenza_rumore
>>> decibel = 10 * math.log10(rapporto)

>>> radianti = 0.7
>>> altezza = math.sin(radianti)
```

Il primo esempio utilizza la funzione `math.log10` per calcolare un rapporto segnale/rumore in decibel (a condizione che siano stati definiti i valori di `potenza_segnale` e `potenza_rumore`). Il modulo math contiene anche `log`, che calcola i logaritmi naturali in base `e`.

Il secondo esempio calcola il seno della variabile `radianti`. Il nome della variabile spiega già che `sin` e le altre funzioni trigonometriche (`cos`, `tan`, ecc.) accettano argomenti espressi in radianti. Per convertire da gradi in radianti occorre dividere per 180 e moltiplicare per π:

```
>>> gradi = 45
>>> radianti = gradi / 180.0 * math.pi
>>> math.sin(radianti)
0.707106781187
```

L'espressione `math.pi` ricava la variabile `pi` dal modulo matematico. Il suo valore è un numero decimale, approssimazione di π, accurata a circa 15 cifre.

Se ricordate la trigonometria, potete verificare il risultato precedente confrontandolo con la radice quadrata di 2 diviso 2:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

Composizione

Finora, abbiamo considerato gli elementi di un programma - variabili, espressioni e istruzioni - separatamente, senza discutere di come utilizzarli insieme.

Una delle caratteristiche più utili dei linguaggi di programmazione è la loro capacità di prendere dei piccoli mattoni e **comporli** tra loro. Per esempio, l'argomento di una funzione può essere un qualunque tipo di espressione, operazioni aritmetiche incluse:

```
x = math.sin(gradi / 360.0 * 2 * math.pi)
```

E anche chiamate di funzione:

```
x = math.exp(math.log(x+1))
```

In linea generale, dovunque potete mettere un valore potete anche mettere un'espressione a piacere, con un'eccezione: il lato sinistro di un'istruzione di assegnazione deve essere un nome di variabile. Ogni altra espressione darebbe un errore di sintassi (vedremo più avanti le eccezioni a questa regola).

```
>>> minuti = ore * 60                # giusto
>>> ore * 60 = minuti                # sbagliato!
SyntaxError: can't assign to operator
```

Aggiungere nuove funzioni

Finora abbiamo usato solo funzioni predefinite o "built-in", che sono parte integrante di Python, ma è anche possibile crearne di nuove. Una **definizione di funzione** specifica il nome di una nuova funzione e la serie di istruzioni che viene eseguita quando la funzione viene chiamata.

Ecco un esempio:

```
def stampa_brani():
    print('Terror di tutta la foresta egli è,')
    print("Con l'ascia in mano si sente un re.")
```

`def` è una parola chiave riservata che indica la definizione di una nuova funzione. Il nome della funzione è `stampa_brani`. Le regole per i nomi delle funzioni sono le stesse dei nomi delle variabili: lettere, numeri e underscore (_) sono permessi, ma il primo carattere non può essere un numero. Non si possono usare parole riservate, e bisogna evitare di avere una funzione e una variabile con lo stesso nome

Le parentesi vuote dopo il nome indicano che la funzione non accetta alcun argomento.

La prima riga della definizione di funzione è chiamata **intestazione**; il resto è detto **corpo**. L'intestazione deve terminare con i due punti, e il corpo deve essere obbligatoriamente indentato, cioè deve avere un rientro rispetto all'intestazione. Per convenzione, l'indentazione è sempre di quattro spazi. Il corpo può contenere un qualsiasi numero di istruzioni.

Le stringhe nelle istruzioni di stampa sono racchiuse tra apici (`' '`) oppure virgolette (`" "`). Virgolette e apici sono equivalenti; la maggioranza degli utenti usa gli apici, eccetto nei casi in cui nel testo da stampare sono contenuti degli apici (che possono essere usati anche come apostrofi o accenti). In questi casi, frequenti con l'italiano, bisogna usare le virgolette.

Virgolette e apici devono essere alti e di tipo indifferenziato, quelli che trovate tra i simboli in alto sulla vostra tastiera. Altre virgolette "tipografiche", come quelle in questa frase, non sono valide in Python.

Se scrivete una funzione in modalità interattiva, l'interprete mette tre puntini di sospensione (...) per indicare che la definizione non è completa:

```
>>> def stampa_brani():
...     print('Terror di tutta la foresta egli è,')
...     print("Con l'ascia in mano si sente un re.")
...
```

Per concludere la funzione, dovete inserire una riga vuota.

La definizione di una funzione crea un **oggetto funzione** che è di tipo `function`:

```
>>> print(stampa_brani)
<function stampa_brani at 0xb7e99e9c>
>>> type(stampa_brani)
<class 'function'>
```

La sintassi per chiamare la nuova funzione è la stessa che abbiamo visto per le funzioni predefinite:

```
>>> stampa_brani()
Terror di tutta la foresta egli è,
Con l'ascia in mano si sente un re.
```

Una volta definita una funzione, si può utilizzarla all'interno di un'altra funzione. Per esempio, per ripetere due volte il brano precedente possiamo scrivere una funzione `ripeti_brani`:

```
def ripeti_brani():
    stampa_brani()
    stampa_brani()
```

E quindi chiamare `ripeti_brani`:

```
>>> ripeti_brani()
Terror di tutta la foresta egli è,
Con l'ascia in mano si sente un re.
Terror di tutta la foresta egli è,
Con l'ascia in mano si sente un re.
```

Ma a dire il vero, la canzone del taglialegna non fa così!

Definizioni e loro utilizzo

Raggruppando assieme i frammenti di codice del Paragrafo precedente, il programma diventa:

```
def stampa_brani():
    print('Terror di tutta la foresta egli è,')
    print("Con l'ascia in mano si sente un re.")

def ripeti_brani():
    stampa_brani()
    stampa_brani()

ripeti_brani()
```

Questo programma contiene due definizioni di funzione: `stampa_brani` e `ripeti_brani`. Le definizioni di funzione sono eseguite come le altre istruzioni, ma il loro effetto è solo quello di creare una nuova funzione. Le istruzioni all'interno di una definizione non vengono eseguite fino a quando la funzione non viene chiamata, e la definizione di per sé non genera alcun risultato.

Ovviamente, una funzione deve essere definita prima di poterla usare: la definizione della funzione deve sempre precedere la sua chiamata.

Come esercizio, spostate l'ultima riga del programma all'inizio, per fare in modo che la chiamata della funzione appaia prima della definizione. Eseguite il programma e guardate che tipo di messaggio d'errore ottenete.

Ora riportate la chiamata della funzione al suo posto, e spostate la definizione di `stampa_brani` dopo la definizione di `ripeti_brani`. Cosa succede quando avviate il programma?

Flusso di esecuzione

Per essere sicuri che una funzione sia stata definita prima di essere utilizzata, dovete conoscere l'ordine in cui le istruzioni vengono eseguite, che è chiamato **flusso di esecuzione**.

L'esecuzione inizia sempre dalla prima istruzione del programma; quindi, le istruzioni successive sono eseguite una alla volta, procedendo dall'alto verso il basso.

Le definizioni di funzione non cambiano il flusso di esecuzione del programma, ma ricordate che le istruzioni all'interno delle funzioni non vengono eseguite fino a quando la funzione non viene chiamata.

Quando viene chiamata una funzione, si genera una specie di deviazione nel flusso di esecuzione: anziché proseguire con l'istruzione successiva, il flusso salta nel corpo della funzione chiamata, ne esegue le istruzioni, e infine riprende il percorso dal punto che aveva lasciato.

Parrebbe tutto abbastanza semplice, se non fosse che una funzione può chiamarne un'altra. Mentre si trova all'interno di una funzione, il programma può dover eseguire le istruzioni che si trovano in un'altra funzione. Poi, mentre esegue quella nuova funzione, il programma potrebbe andare ad eseguirne un'altra ancora!

Fortunatamente, Python sa tener bene traccia di dove si trova: ogni volta che una funzione viene completata, il programma ritorna al punto della funzione chiamante che aveva lasciato. E una volta giunto alla fine, termina il suo lavoro.

Concludendo, nel leggere un programma non è sempre opportuno farlo dall'alto in basso. Spesso è più logico seguire il flusso di esecuzione.

Parametri e argomenti

Alcune delle funzioni che abbiamo visto richiedono degli argomenti. Per esempio, se volete trovare il seno di un numero chiamando la funzione `math.sin`, dovete passarle quel numero come argomento. Alcune funzioni ricevono più di un argomento: a `math.pow` ne servono due, che sono la base e l'esponente dell'operazione di elevamento a potenza.

All'interno della funzione, gli argomenti che le vengono passati sono assegnati ad altrettante variabili chiamate **parametri**. Ecco un esempio di definizione di una funzione che riceve un argomento:

```
def stampa2volte(bruce):
    print(bruce)
    print(bruce)
```

Questa funzione assegna l'argomento ricevuto ad un parametro chiamato `bruce`. Quando la funzione viene chiamata, stampa il valore del parametro (qualunque esso sia) due volte.

Questa funzione elabora qualunque valore che possa essere stampato.

```
>>> stampa2volte('Spam')
Spam
Spam
>>> stampa2volte(42)
42
42
>>> stampa2volte(math.pi)
3.14159265359
3.14159265359
```

Le stesse regole di composizione che valgono per le funzioni predefinite si applicano anche alle funzioni definite da un programmatore, pertanto possiamo usare come argomento per `stampa2volte` qualsiasi espressione:

```
>>> stampa2volte('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> stampa2volte(math.cos(math.pi))
-1.0
-1.0
```

L'argomento viene valutato prima della chiamata alla funzione, pertanto nell'esempio appena proposto le espressioni `'Spam '*4` e `math.cos(math.pi)` vengono valutate una volta sola.

Potete anche usare una variabile come argomento di una funzione:

```
>>> michael = 'Eric, the half a bee.'
>>> stampa2volte(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Il nome della variabile che passiamo come argomento (`michael`) non ha niente a che fare con il nome del parametro nella definizione della funzione (`bruce`). Non ha importanza come era stato denominato il valore di partenza (nel codice chiamante); qui in `stampa2volte`, chiamiamo tutto quanto `bruce`.

Variabili e parametri sono locali

Quando create una variabile in una funzione, essa è **locale**, cioè esiste solo all'interno della funzione. Per esempio:

```
def cat2volte(parte1, parte2):
    cat = parte1 + parte2
    stampa2volte(cat)
```

Questa funzione prende due argomenti, li concatena e poi stampa il risultato per due volte. Ecco un esempio che la utilizza:

```
>>> riga1 = 'Bing tiddle '
>>> riga2 = 'tiddle bang.'
>>> cat2volte(riga1, riga2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Quando `cat2volte` termina, la variabile `cat` viene distrutta. Se provassimo a stamparla, otterremmo infatti un messaggio d'errore:

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Anche i parametri sono locali: esternamente alla funzione `stampa2volte`, non esiste nulla di nome `bruce`.

Funzioni "produttive" e funzioni "vuote"

Alcune delle funzioni che abbiamo usato, come le funzioni matematiche, restituiscono dei risultati; in mancanza di definizioni migliori, personalmente le chiamo **funzioni "produttive"**. Altre funzioni, come `stampa2volte`, eseguono un'azione ma non restituiscono alcun valore. Le chiameremo **funzioni "vuote"**.

Quando chiamate una funzione produttiva, quasi sempre è per fare qualcosa di utile con il suo risultato, tipo assegnarlo a una variabile o usarlo come parte di un'espressione.

```
x = math.cos(radianti)
aureo = (math.sqrt(5) + 1) / 2
```

Se chiamate una funzione in modalità interattiva, Python ne mostra il risultato:

```
>>> math.sqrt(5)
2.2360679774997898
```

Ma in uno script, se chiamate una funzione produttiva così come è, il valore di ritorno è perso!

```
math.sqrt(5)
```

Questo script in effetti calcola la radice quadrata di 5, ma non conserva nè visualizza il risultato, per cui non è di grande utilità.

Le funzioni vuote possono visualizzare qualcosa sullo schermo o avere qualche altro effetto, ma non restituiscono un valore. Se provate comunque ad assegnare il risultato ad una variabile, ottenete un valore speciale chiamato `None` (nulla).

```
>>> risultato = stampa2volte('Bing')
Bing
Bing
>>> print(risultato)
None
```

Il valore `None` non è la stessa cosa della stringa `'None'`. È un valore speciale che appartiene ad un tipo tutto suo:

```
>>> type(None)
<class 'NoneType'>
```

Le funzioni che abbiamo scritto finora, sono tutte vuote. Cominceremo a scriverne di produttive tra alcuni capitoli.

Debug

Saper rintracciare e correggere gli errori è una essenziale qualità che dovete acquisire. Anche se a volte può essere demotivante, si tratta infatti di una delle parti più intellettualmente ricche, stimolanti ed interessanti della programmazione.

Possiamo paragonare il debug al lavoro di un investigatore: avete a disposizione degli indizi e dovete ricostruire quali processi ed eventi hanno prodotto il risultato che osservate.

Il debug è anche simile ad una scienza sperimentale. Quando pensate di aver capito cosa può avere provocato un errore, modificate il programma di conseguenza e riprovate di nuovo. Se l'ipotesi era giusta,

avete saputo prevedere il risultato della modifica e vi siete avvicinati di un passo ad un programma funzionante. Se l'ipotesi era sbagliata, ne dovete formulare un'altra. Come disse Sherlock Holmes: "Una volta eliminato l'impossibile, qualsiasi cosa rimanga, per quanto improbabile, deve essere la verità." (A. Conan Doyle, *Il segno dei quattro*)

Per alcuni, la programmazione e la rimozione degli errori sono in fondo la stessa cosa: programmare è una procedura di graduale rimozione degli errori da un programma, fino a quando non funziona a dovere. L'idea di fondo è di iniziare con un programma funzionante e di fare ogni volta piccole modifiche, effettuandone man mano il debug.

Linux, ad esempio, è un sistema operativo fatto da milioni di righe di codice, ma nacque come un semplice programma che Linus Torvalds usava per esplorare il chip Intel 80386. Secondo Larry Greenfields, "Uno dei progetti iniziali di Linus era un programma che doveva visualizzare alternativamente una sequenza di AAAA e BBBB. Questo programma si è poi evoluto in Linux". (*The Linux Users' Guide* Beta Version 1).

Glossario

funzione: Una serie di istruzioni dotata di un nome che esegue una certa operazione utile. Le funzioni possono o meno ricevere argomenti e possono o meno produrre un risultato.

definizione di funzione: Istruzione che crea una nuova funzione, specificandone il nome, i parametri, e le istruzioni che contiene.

oggetto funzione: Valore creato da una definizione di funzione. Il nome della funzione è una variabile che fa riferimento a un oggetto funzione.

intestazione: La prima riga di una definizione di funzione.

corpo: La serie di istruzioni all'interno di una definizione di funzione.

parametro: Un nome usato all'interno di una funzione che fa riferimento al valore passato come argomento.

chiamata di funzione: Istruzione che esegue una funzione. Consiste nel nome della funzione seguito da un elenco di argomenti tra parentesi.

argomento: Un valore fornito (passato) a una funzione quando viene chiamata. Questo valore viene assegnato al corrispondente parametro nella funzione.

variabile locale: Variabile definita all'interno di una funzione e che puo essere usata solo all'interno della funzione.

valore di ritorno: Il risultato di una funzione. Se una chiamata di funzione viene usata come espressione, il valore di ritorno è il valore dell'espressione.

funzione "produttiva": Una funzione che restituisce un valore.

funzione "vuota": Una funzione che restituisce sempre `None`.

`None`**:** Valore speciale restituito dalle funzioni vuote.

modulo: Un file che contiene una raccolta di funzioni correlate e altre definizioni.

istruzione import: Istruzione che legge un file modulo e crea un oggetto modulo utilizzabile.

oggetto modulo: Valore creato da un'istruzione `import` che fornisce l'accesso ai valori definiti in un modulo.

dot notation o notazione a punto: Sintassi per chiamare una funzione di un modulo diverso, specificando il nome del modulo seguito da un punto e dal nome della funzione.

composizione: Utilizzare un'espressione come parte di un'espressione più grande o un'istruzione come parte di un'istruzione più grande.

flusso di esecuzione: L'ordine in cui vengono eseguite le istruzioni nel corso di un programma.

Esercitazione: Progettazione dell'interfaccia

Questa esercitazione prosegue con l'uso del modulo `turtle` per dimostrare una procedura per progettare delle funzioni che collaborano tra loro.

Ripetizione semplice

Create un file di nome `miopoligono.py` e scriveteci il seguente codice, che disegna un quadrato sullo schermo:

```
import turtle
bob = turtle.Turtle()

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
turtle.mainloop()
```

Si può ottenere lo stesso risultato in modo più conciso con un'istruzione `for`. Aggiungete questo esempio a `miopoligono.py` ed eseguitelo di nuovo:

```
for i in range(4):
    print('Ciao!')
```

Dovreste vedere qualcosa di simile:

```
Ciao!
Ciao!
Ciao!
Ciao!
```

Questo è l'utilizzo più semplice dell'istruzione `for`; ne esistono altri più sofisticati. Ma questo dovrebbe bastare per permettervi di riscrivere il programma di disegno di quadrati. Proseguite nella prossima pagina solo dopo averlo fatto.

Questa esercitazione è un adattamento del Capitolo 4 di "Pensare in Python" di A. Downey, traduzione di A. Zanella. `https://github.com/AllenDowney/ThinkPythonItalian`

Ecco l'istruzione `for` che disegna un quadrato:

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

La sintassi di un'istruzione `for` è simile a quella di una funzione. Ha un'intestazione che termina con i due punti e un corpo indentato che può contenere un numero qualunque di istruzioni.

Un'istruzione `for` è chiamata anche **ciclo**, perché il flusso dell'esecuzione ne attraversa il corpo per poi ritornare indietro e ripeterlo da capo. In questo caso, il corpo viene eseguito per quattro volte.

Questa versione del disegno di quadrati è in realtà un pochino differente dalla precedente, in quanto provoca un'ultima svolta dopo aver disegnato l'ultimo lato. Ciò comporta del tempo in più, ma il codice viene semplificato, inoltre lascia la tartaruga nella stessa posizione di partenza, rivolta nella direzione iniziale.

Esercizi

Quella che segue è una serie di esercizi che utilizzano `turtle`. Sono pensati per essere divertenti, ma hanno anche uno scopo. Mentre ci lavorate su, provate a pensare quale sia.

1. Scrivete una funzione di nome `quadrato` che richieda un parametro di nome `t`, che è una tartaruga. La funzione deve usare la tartaruga per disegnare un quadrato.

   Scrivete una chiamata alla funzione `quadrato` che passi `bob` come argomento, ed eseguite nuovamente il programma.

2. Aggiungete a `quadrato` un nuovo parametro di nome `lunghezza`. Modificate il corpo in modo che la lunghezza dei lati sia pari a `lunghezza`, quindi modificate la chiamata alla funzione in modo da fornire un secondo argomento. Eseguite di nuovo il programma e provatelo con vari valori di `lunghezza`.

3. Fate una copia di `quadrato` e cambiate il nome in `poligono`. Aggiungete un altro parametro di nome `n` e modificate il corpo in modo che sia disegnato un poligono regolare di n lati. Suggerimento: gli angoli esterni di un poligono regolare di n lati misurano $360/n$ gradi.

4. Scrivete una funzione di nome `cerchio` che prenda come parametri una tartaruga, `t`, e un raggio, `r`, e che disegni un cerchio approssimato chiamando `poligono` con un'appropriata lunghezza e numero di lati. Provate la funzione con diversi valori di `r`.

   Suggerimento: pensate alla circonferenza del cerchio e accertatevi che `lunghezza * n = circonferenza`.

5. Create una versione più generale della funzione `cerchio`, di nome `arco`, che richieda un parametro aggiuntivo `angolo`, il quale determina la porzione di cerchio da disegnare. `angolo` è espresso in gradi, quindi se `angolo=360`, `arco` deve disegnare un cerchio completo.

Soluzioni: Progettazione dell'interfaccia

Incapsulamento

Il primo esercizio chiede di inserire il codice per disegnare un quadrato in una definizione di funzione, passando la tartaruga come argomento. Ecco una soluzione:

```
def quadrato(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

quadrato(bob)
```

Le istruzioni più interne, `fd` e `lt` sono doppiamente indentate per significare che si trovano all'interno del ciclo `for`, che a sua volta è all'interno della funzione. L'ultima riga, `quadrato(bob)`, è a livello del margine sinistro, pertanto indica la fine sia del ciclo `for` che della definizione di funzione.

Dentro la funzione, `t` si riferisce alla stessa tartaruga a cui si riferisce `bob`, per cui `t.lt(90)` ha lo stesso effetto di `bob.lt(90)`. Ma allora perché non chiamare `bob` il parametro? Il motivo è che `t` può essere qualunque tartaruga, non solo `bob`, e in questa maniera è possibile anche creare una seconda tartaruga e passarla come parametro a `quadrato`:

```
alice = turtle.Turtle()
quadrato(alice)
```

L'inglobare un pezzo di codice in una funzione è chiamato **incapsulamento**. Uno dei benefici dell'incapsulamento è che appiccica un nome al codice, il che può servire come una sorta di documentazione. Un altro vantaggio è il riuso del codice: è più conciso chiamare una funzione due volte che copiare e incollare il corpo!

Generalizzazione

Il passo successivo è aggiungere a `quadrato` un parametro `lunghezza`. Ecco una soluzione:

```
def quadrato(t, lunghezza):
    for i in range(4):
        t.fd(lunghezza)
        t.lt(90)

quadrato(bob, 100)
```

L'aggiunta di un parametro a una funzione è chiamata **generalizzazione** poiché rende la funzione più generale: nella versione precedente, il quadrato aveva sempre la stessa dimensione, ora può essere grande a piacere.

Anche il passo seguente è una generalizzazione. Invece di disegnare solo quadrati, `poligono` disegna poligoni regolari di un qualunque numero di lati. Ecco una soluzione:

```
def poligono(t, n, lunghezza):
    angolo = 360 / n
    for i in range(n):
        t.fd(lunghezza)
        t.lt(angolo)

poligono(bob, 7, 70)
```

Questo esempio disegna un ettagono regolare con lati di lunghezza 70.

Quando in una chiamata di funzione avete più di qualche argomento numerico, è facile dimenticare a cosa si riferiscono o in quale ordine vanno disposti. In questi casi, è bene includere i nomi dei parametri nell'elenco degli argomenti:

```
poligono(bob, n=7, lunghezza=70)
```

Questi sono detti **argomenti con nome** perché includono il nome del parametro a cui vengono passati, quale "parola chiave" (da non confondere con le parole chiave riservate come `while` e `def`).

Questa sintassi rende il programma più leggibile. È anche un appunto di come funzionano argomenti e parametri: quando chiamate una funzione, gli argomenti vengono assegnati a quei dati parametri.

Progettazione dell'interfaccia

Il prossimo passaggio è scrivere `cerchio`, che richiede come parametro il raggio, `r`. Ecco una semplice soluzione che usa `poligono` per disegnare un poligono di 50 lati:

```
import math

def cerchio(t, r):
    circonferenza = 2 * math.pi * r
    n = 50
    lunghezza = circonferenza / n
    poligono(t, n, lunghezza)
```

La prima riga calcola la circonferenza di un cerchio di raggio `r` usando la nota formula $2\pi r$. Dato che usiamo `math.pi`, vi ricordo che dovete prima importare il modulo `math`. Per convenzione, l'istruzione `import` si scrive all'inizio dello script.

`n` è il numero di segmenti del nostro cerchio approssimato, e `lunghezza` è la lunghezza di ciascun segmento. Così facendo, `poligono` disegna un poligono di 50 lati che approssima un cerchio di raggio `r`.

Un limite di questa soluzione è che `n` è costante, il che comporta che per cerchi molto grandi i segmenti sono troppo lunghi, e per cerchi piccoli perdiamo tempo a disegnare minuscoli segmenti. Una soluzione sarebbe di generalizzare la funzione tramite un parametro `n`, dando all'utente (chiunque chiami la funzione `cerchio`) più controllo, ma rendendo così l'interfaccia meno chiara.

L'**interfaccia** è un riassunto di come è usata la funzione: quali sono i parametri? Che cosa fa la funzione? Qual è il valore restituito? Un'interfaccia è considerata "pulita" se permette al chiamante di fare ciò che deve, senza avere a che fare con dettagli non necessari.

In questo esempio, `r` appartiene all'interfaccia perché specifica il cerchio da disegnare. `n` è meno pertinente perché riguarda i dettagli di *come* il cerchio viene reso.

Piuttosto di ingombrare l'interfaccia di parametri, è meglio scegliere un valore appropriato di `n` che dipenda da `circonferenza`:

```
def cerchio(t, r):
    circonferenza = 2 * math.pi * r
    n = int(circonferenza / 3) + 3
    lunghezza = circonferenza / n
    poligono(t, n, lunghezza)
```

Ora il numero di segmenti è un numero intero vicino a `circonferenza/3`, e la lunghezza dei segmenti è circa 3, che è abbastanza piccolo da dare un cerchio di bell'aspetto, ma abbastanza grande da essere efficiente e appropriato per qualsiasi dimensione del cerchio.

Aggiungere 3 a `n` garantisce che il poligono abbia come minimo 3 lati.

Refactoring

Nello scrivere `cerchio`, ho potuto riusare `poligono` perché un poligono con molti lati è una buona approssimazione di un cerchio. Ma la funzione `arco` non è così collaborativa: non possiamo usare `poligono` o `cerchio` per disegnare un arco.

Un'alternativa è partire da una copia di `poligono` e trasformarla in `arco`. Il risultato può essere qualcosa del genere:

```
def arco(t, r, angolo):
    arco_lunghezza = 2 * math.pi * r * angolo / 360
    n = int(arco_lunghezza / 3) + 1
    passo_lunghezza = arco_lunghezza / n
    passo_angolo = angolo / n

    for i in range(n):
        t.fd(passo_lunghezza)
        t.lt(passo_angolo)
```

La seconda metà di questa funzione somiglia a `poligono`, ma non possiamo riusare questa funzione senza cambiarne l'interfaccia. Potremmo generalizzare `poligono` in modo che riceva un angolo come terzo argomento, ma allora `poligono` non sarebbe più un nome appropriato! Invece, creiamo una funzione più generale chiamata `polilinea`:

```
def polilinea(t, n, lunghezza, angolo):
    for i in range(n):
        t.fd(lunghezza)
        t.lt(angolo)
```

Ora possiamo riscrivere `poligono` e `arco` in modo che usino `polilinea`:

```
def poligono(t, n, lunghezza):
    angolo = 360.0 / n
    polilinea(t, n, lunghezza, angolo)

def arco(t, r, angolo):
    arco_lunghezza = 2 * math.pi * r * angolo / 360
    n = int(arco_lunghezza / 3) + 1
    passo_lunghezza = arco_lunghezza / n
    passo_angolo = float(angolo) / n
    polilinea(t, n, passo_lunghezza, passo_angolo)
```

Infine, riscriviamo `cerchio` in modo che usi `arco`:

```
def cerchio(t, r):
    arco(t, r, 360)
```

Questo procedimento di riarrangiare una programma per migliorare le interfacce e facilitare il riuso del codice, è chiamato **refactoring**. In questo caso, abbiamo notato che in `arco` e in `poligono` c'era del codice simile, allora abbiamo semplificato il tutto in `polilinea`.

Avendoci pensato prima, avremmo potuto scrivere `polilinea` direttamente, evitando il refactoring, ma spesso all'inizio di un lavoro non si hanno le idee abbastanza chiare per progettare al meglio tutte le interfacce. Una volta cominciato a scrivere il codice, si colgono meglio i problemi. A volte, il refactoring è segno che avete imparato qualcosa.

Tecnica di sviluppo

Una **tecnica di sviluppo** è una procedura di scrittura dei programmi. Quello che abbiamo usato in questa esercitazione si chiama "incapsulamento e generalizzazione". I passi della procedura sono:

1. Iniziare scrivendo un piccolo programma senza definire funzioni.
2. Una volta ottenuto un programma funzionante, identificare una sua porzione che sia in sé coerente e autonoma, incapsularla in una funzione e dargli un nome.
3. Generalizzare la funzione aggiungendo i parametri appropriati.
4. Ripetere i passi da 1 a 3 fino ad avere un insieme di funzioni. Copiate e incollate il codice funzionante per evitare di riscriverlo (e ricorreggerlo).
5. Cercare le occasioni per migliorare il programma con il refactoring. Ad esempio, se avete del codice simile in più punti, valutate di semplificare rielaborandolo in una funzione più generale.

Questa procedura ha alcuni inconvenienti—vedremo più avanti alcune alternative—ma può essere di aiuto se in principio non sapete bene come suddividere il vostro programma in funzioni. È un approccio che vi permette di progettare man mano che andate avanti.

Stringa di documentazione

Una **stringa di documentazione**, o *docstring*, è una stringa posta all'inizio di una funzione che ne illustra l'interfaccia. Ecco un esempio:

```
def polilinea(t, n, lunghezza, angolo):
    """Disegna n segmenti di data lunghezza e angolo
       (in gradi) tra di loro. t e' una tartaruga.
    """
    for i in range(n):
        t.fd(lunghezza)
        t.lt(angolo)
```

Per convenzione, la docstring è racchiusa tra triple virgolette, che le consentono di essere divisibile su più righe (stringa a righe multiple).

È breve, ma contiene le informazioni essenziali di cui qualcuno potrebbe aver bisogno per usare la funzione. Spiega in modo conciso cosa fa la funzione (senza entrare nei dettagli di come lo fa). Spiega che effetti ha ciascun parametro sul comportamento della funzione e di che tipo devono essere i parametri stessi (se non è ovvio).

Scrivere questo tipo di documentazione è una parte importante della progettazione dell'interfaccia. Un'interfaccia ben studiata dovrebbe essere semplice da spiegare; se fate fatica a spiegare una delle vostre funzioni, può darsi che la sua interfaccia sia migliorabile.

Debug

Un'interfaccia è simile ad un contratto tra la funzione e il suo chiamante. Il chiamante si impegna a fornire certi parametri e la funzione si impegna a svolgere un dato lavoro.

Ad esempio, a `polilinea` devono essere passati quattro argomenti: `t` deve essere una tartaruga; `n` deve essere un numero intero; `lunghezza` deve essere un numero positivo; e `angolo` un numero che si intende espresso in gradi.

Questi requisiti sono detti **precondizioni** perché si suppone siano verificati prima che la funzione sia eseguita. Per contro, le condizioni che si devono verificare al termine della funzione sono dette **postcondizioni**, e comprendono l'effetto che deve avere la funzione (come il disegnare segmenti) e ogni altro effetto minore (come muovere la tartaruga o fare altri cambiamenti).

Le precondizioni sono responsabilità del chiamante. Se questi viola una precondizione (documentata in modo appropriato!) e la funzione non fa correttamente ciò che deve, l'errore sta nel chiamante e non nella funzione.

Se le precondizioni sono soddisfatte e le postcondizioni no, l'errore sta nella funzione. E il fatto che le vostre pre- e postcondizioni siano chiare, è di aiuto nel debug.

Glossario

metodo: Una funzione associata ad un oggetto che viene chiamata utlizzando la notazione a punto.

ciclo: Una porzione di programma che può essere eseguita ripetutamente.

incapsulamento: Il procedimento di trasformare una serie di istruzioni in una funzione.

generalizzazione: Il procedimento di sostituire qualcosa di inutilmente specifico (come un numero) con qualcosa di più generale ed appropriato (come una variabile o un parametro).

argomento con nome: Un argomento che include il nome del parametro a cui è destinato come "parola chiave".

interfaccia: Una descrizione di come si usa una funzione, incluso il nome, la descrizione degli argomenti e il valore di ritorno.

refactoring: Il procedimento di modifica di un programma funzionante per migliorare le interfacce delle funzioni e altre qualità del codice.

tecnica di sviluppo: Procedura di scrittura dei programmi.

stringa di documentazione o docstring: Una stringa che compare all'inizio di una definizione di una funzione per documentarne l'interfaccia.

precondizione: Un requisito che deve essere soddisfatto dal chiamante prima di eseguire una funzione.

postcondizione: Un requisito che deve essere soddisfatto dalla funzione prima di terminare.