Corso di programmazione in Python

Lezione 3

Neapolis Hacklab

hacklab@officina99.org

Lezione 3 - Argomenti

- Introduzione alla Python Stardard Library
- Basi di I/O
- Gestione delle eccezioni

Organizzazione del codice

L'organizzazione del codice in Python avviene principalmente attraverso i *moduli*.

Essi sono semplici file, con estenzione *.py*, che contengono codice Python.

Lo scopo, come per qualsiasi altro linguaggio, è di dividere l'applicazione in più file e di riutilizzare codice generico.

Organizzazione del codice

L'organizzazione del codice in Python avviene principalmente attraverso i *moduli*.

Essi sono semplici file, con estenzione *.py*, che contengono codice Python.

Lo scopo, come per qualsiasi altro linguaggio, è di dividere l'applicazione in più file e di riutilizzare codice generico.

In Python ogni file (o modulo) deve *importare* esplicitamente le funzionalità di altri moduli di cui necessita.

In questo modo è molto più semplice conoscere le connessioni tra i vari moduli dell'applicazione e quindi anche quale impatto potranno avere eventuali modifiche ad un modulo.

Organizzazione del codice

Il comando responsabile dell'importazione di funzionalità da un modulo è `import` .
Un esempio seguente chiarirà il funzionamento dell'importazione.

Organizzazione del codice

Il comando responsabile dell'importazione di funzionalità da un modulo è `import` .
Un esempio seguente chiarirà il funzionamento dell'importazione.

Supponiamo di avere due file nella nostra directory di sviluppo, il primo *main.py* il file principale dell'applicazione e il secondo *utility.py* contenente delle funzioni di utilità.

Definiamo per esempio due funzioni nel file utility.py chiamate *chiedi* e *chiediNome* che visualizzano un messaggio, attendono un input da tastiera da parte dell'utente e lo restituiscono.

Organizzazione del codice

Utility.py

```
1  def chiedi(domanda):
2      return raw_input(domanda + " > ")
3
4  def chiediNome():
5      return chiedi("Nome")
```

Organizzazione del codice

Utility.py

```
1  def chiedi(domanda):
2      return raw_input(domanda + " > ")
3
4  def chiediNome():
5      return chiedi("Nome")
```

Vediamo come importare queste funzioni in main.py .

Organizzazione del codice

Vediamo di seguito il file *main.py* che utilizza le funzioni del modulo *utility*.

```
1  import utility
2
3  nome = utility.chiediNome()
4  print "Ciao %s!" % nome
```

Riga 1: viene importato il modulo *utility*. Da questo punto in poi tutte le funzioni, classi e variabili definite nel modulo *utility* saranno disponibili.

Organizzazione del codice

Vediamo di seguito il file *main.py* che utilizza le funzioni del modulo *utility*.

```
1  import utility
2
3  nome = utility.chiediNome()
4  print "Ciao %s!" % nome
```

Riga 1: viene importato il modulo *utility*. Da questo punto in poi tutte le funzioni, classi e variabili definite nel modulo *utility* saranno disponibili.

Riga 3: la funzione *chiediNome* viene chiamata specificando il nome del modulo *utility*.
Anche i moduli sono oggetti: il loro contenuto definisce i loro attributi.

Organizzazione del codice

L'importazione del modulo *utility* come visto nella riga 1 rende disponibili al modulo in cui viene importato tutto ciò che contiene: variabili, funzioni e classi, attraverso l'utilizzo del nome del modulo *utility*.

Organizzazione del codice

L'importazione del modulo *utility* come visto nella riga 1 rende disponibili al modulo in cui viene importato tutto ciò che contiene: variabili, funzioni e classi, attraverso l'utilizzo del nome del modulo *utility*.

É possibile anche specificare precisamente cosa voler importare di uno specifico modulo nel seguente modo:

Organizzazione del codice

L'importazione del modulo *utility* come visto nella riga 1 rende disponibili al modulo in cui viene importato tutto ciò che contiene: variabili, funzioni e classi, attraverso l'utilizzo del nome del modulo *utility*.

É possibile anche specificare precisamente cosa voler importare di uno specifico modulo nel seguente modo:

```
1  from utility import chiediNome
2
3  nome = chiediNome()
4  print "Ciao %s!" % nome
```

Organizzazione del codice

L'importazione del modulo *utility* come visto nella riga 1 rende disponibili al modulo in cui viene importato tutto ciò che contiene: variabili, funzioni e classi, attraverso l'utilizzo del nome del modulo *utility*.

É possibile anche specificare precisamente cosa voler importare di uno specifico modulo nel seguente modo:

```
1  from utility import chiediNome
2
3  nome = chiediNome()
4  print "Ciao %s!" % nome
```

In questo modo importiamo direttamente la funzione *chiediNome*, rendendola accessibile come se fosse stata definita nello stesso modulo che la utilizza.

Organizzazione del codice

É possibile inoltre specificare l'inclusione di tutto ciò che è contenuto in un modulo:

```
1  from utility import *
2
3  nome = chiediNome()
4  print "Ciao %s!" % nome
5
6  cognome = chiedi("Cognome")
7  print "Il tuo cognome: %s" % cognome
```

rendendo quindi ogni cosa definita nel modulo *utility* disponibile direttamente nel modulo che importa.

Organizzazione del codice

Come abbiamo visto esistono più modi per importare funzionalità da altri moduli.

Affinchè possa essere semplice gestire molto codice diviso in vari moduli è buona norma importare solo il necessario dai moduli che si utilizza.

Con semplici regole possiamo scegliere l'inclusione migliore per le nostre esigenze e quindi tenere sotto controllo le dipendeze di ogni modulo e semplificare la ricerca di eventuali bug.

Organizzazione del codice

Nel caso in cui dobbiamo importare poche funzionalità di un modulo, rispetto a quante ne possiede è meglio utilizzare l'inclusione specifica.
Ovvero specificando precisamente quali classi, funzioni o variabili dobbiamo includere:

```
1  from utility import chiediNome
2
3  nome = chiediNome()
4  print "Ciao %s!" % nome
```

Organizzazione del codice

Nel caso in cui le funzionalità di cui si necessità sono la maggior parte di quelle definite nel modulo da importare è bene importare direttamente il modulo intero, come visto in precedenza nell'esempio:

```
1  import utility
2
3  nome = utility.chiediNome()
4  print "Ciao %s!" % nome
```

Organizzazione del codice

Ovviamente ciò è possibile anche con l'inclusione totale di tutte le funzionalità come visto in precedenza nell'esempio:

```
1  from utility import *
2
3  nome = chiediNome()
4  print "Ciao %s!" % nome
5
6  cognome = chiedi("Cognome")
7  print "Il tuo cognome: %s" % cognome
```

Organizzazione del codice

Ovviamente ciò è possibile anche con l'inclusione totale di tutte le funzionalità come visto in precedenza nell'esempio:

```
1  from utility import *
2
3  nome = chiediNome()
4  print "Ciao %s!" % nome
5
6  cognome = chiedi("Cognome")
7  print "Il tuo cognome: %s" % cognome
```

Fate attenzione però ad usare questa modalità di inclusione. É necessario che si conosca benissimo il modulo che si sta includendo. Tutte le funzioni, classi e variabili saranno incluse direttamente rischiando di avere conflitti tra i nomi e quindi sovrascrivendo gli oggetti definiti nel modulo che si include.

Organizzazione del codice

Nel caso in cui anche il nome di un modulo può causare conflitto, Python permette di creare un alias per il modulo che si sta includendo.
Ciò funziona anche per i singoli oggetti che si importano da un modulo.

```
1  import utility as ut
2  nome = ut.chiediNome()
3
4  from utility import chiediNome as cn
5  nome = cn()
```

Il risultato delle prime due righe è lo stesso delle altre due righe di codice. La differenza è nel modo di inclusione.

Organizzazione del codice

Nel caso in cui anche il nome di un modulo può causare conflitto, Python permette di creare un alias per il modulo che si sta includendo.
Ciò funziona anche per i singoli oggetti che si importano da un modulo.

```
1  import utility as ut
2  nome = ut.chiediNome()
3
4  from utility import chiediNome as cn
5  nome = cn()
```

Nel primo caso si importa il modulo *utility* creando l'alias `ut` dal quale si accederà alle funzionalità del modulo.

Organizzazione del codice

Nel caso in cui anche il nome di un modulo può causare conflitto, Python permette di creare un alias per il modulo che si sta includendo.
Ciò funziona anche per i singoli oggetti che si importano da un modulo.

```
1  import utility as ut
2  nome = ut.chiediNome()
3
4  from utility import chiediNome as cn
5  nome = cn()
```

Nel secondo caso si importa solo la funzione *chiediNome* creando un alias `cn` che vi farà riferimento.

Organizzazione del codice

I moduli a loro volta possono essere raggruppati in *Package*. Essi non sono altro che semplici directory contenenti moduli python.

Organizzazione del codice

I moduli a loro volta possono essere raggruppati in *Package*. Essi non sono altro che semplici directory contenenti moduli python.

Supponiamo di voler raggruppare una serie di moduli in due package come si seguito:

- hardware
  - video
  - audio
  - input
- logic
  - sprite
  - surface
  - engine

Organizzazione del codice

Per creare i due package *hardware* e *login* bisogna creare due directory con i nomi dei package.
Ognuna di esse dovrà contenere i relativi moduli, ovvero i file con codice Python con estensione .py .

Organizzazione del codice

Per creare i due package *hardware* e *login* bisogna creare due directory con i nomi dei package.
Ognuna di esse dovrà contenere i relativi moduli, ovvero i file con codice Python con estensione .py .

Quindi creiamo all'interno della dir *hardware* i file *video.py*, *audio.py* e *input.py*.

Organizzazione del codice

Per creare i due package *hardware* e *login* bisogna creare due directory con i nomi dei package.
Ognuna di esse dovrà contenere i relativi moduli, ovvero i file con codice Python con estensione .py .

Quindi creiamo all'interno della dir *hardware* i file *video.py*, *audio.py* e *input.py*.

Lo stesso per il package *logic*. Creiamo la directory *logic*, e al suo interno creiamo i file *sprite.py*, *surface.py* e *engine.py*.

Organizzazione del codice

Bisogna ora indicare a Python che quelle due directory sono package e che quindi quando importiamo funzioni da un modulo bisogna anche ricercare in queste directory.

Organizzazione del codice

Bisogna ora indicare a Python che quelle due directory sono package e che quindi quando importiamo funzioni da un modulo bisogna anche ricercare in queste directory.

Ciò avviene creando un file di nome *__init__.py* all'interno di ognuna delle directory.

Organizzazione del codice

Bisogna ora indicare a Python che quelle due directory sono package e che quindi quando importiamo funzioni da un modulo bisogna anche ricercare in queste directory.

Ciò avviene creando un file di nome *__init__.py* all'interno di ognuna delle directory.

Questi file in genere sono vuoti, ma possono comunque contenere il codice Python di inizializzazione del package, ovvero quel codice che viene eseguito quando viene importato il package e che serve al funzionamento di tutti i moduli al suo interno.

Organizzazione del codice

Se questi due package appena creati si trovano nella nostra directory di sviluppo e supponendo di aver scritto del codice in ognuno dei moduli, possiamo importarne le funzionalità nello stesso modo visto in precedenza.

Organizzazione del codice

Se questi due package appena creati si trovano nella nostra directory di sviluppo e supponendo di aver scritto del codice in ognuno dei moduli, possiamo importarne le funzionalità nello stesso modo visto in precedenza.

Ad esempio:

```
1  import hardware.video
2  hardware.video.inizializza(640, 480)
3
4  from hardware import audio
5  audio.set_volume(50)
6
7  from logic.engine import Cube3D
8  c = Cube3D()
```

Organizzazione del codice

```
1  import hardware.video
2  hardware.video.inizializza(640, 480)
3
4  from hardware import audio
5  audio.set_volume(50)
6
7  from logic.engine import Cube3D
8  c = Cube3D()
```

Riga 1: importa il modulo *video* dal package *hardware*.
Riga 4: importa il modulo *audio* dal package *hardware*.
Riga 7: importa la classe *Cube3D* dal modulo *engine* del package *logic*.

Organizzazione del codice

Resta ora da chiarire un ultimo punto: come fa Python a sapere dove cercare i moduli da importare?

Organizzazione del codice

Resta ora da chiarire un ultimo punto: come fa Python a sapere dove cercare i moduli da importare?

Innanzitutto *Python cerca i moduli nella directory principale del programma*, ovvero dov'è contenuto il file principale che viene passato all'interprete di Python per l'esecuzione. Ecco perchè era necessario che i due package creati prima si trovassero nella directory di sviluppo.

Organizzazione del codice

Resta ora da chiarire un ultimo punto: come fa Python a sapere dove cercare i moduli da importare?

Innanzitutto *Python cerca i moduli nella directory principale del programma*, ovvero dov'è contenuto il file principale che viene passato all'interprete di Python per l'esecuzione. Ecco perchè era necessario che i due package creati prima si trovassero nella directory di sviluppo.

Se il modulo da importare non viene trovato Python continua a cercare nelle directory specificate nella variabile d'ambiente *PYTHONPATH*. E' una semplice lista di directory separata da *':'* (due punti) .

Organizzazione del codice

Se il modulo ancora non è stato trovato Python conclude la ricerca nelle proprie directory di sistema, ovvero quelle create al momento dell' installazione di Python.
Esse cambiano in base al sistema operativo su cui vengono installate.
Supponendo di aver installato Python 2.5 su Linux la ricerca si concluderebbe nelle directory:

- */usr/lib/python2.5*, che contiene la Python Stardard Library.
- */usr/lib/python2.5/site-packages*, che contiene i package di terze parti, ovvero quelli installati nel sistema che non fanno parte della standard library.

Basi di I/O

Le precedenti slide sull'organizzazione del codice ci serviranno a capire meglio la struttura della Python Standard Library (da ora *PSL*) e come utilizzarla.

Basi di I/O

Le precedenti slide sull'organizzazione del codice ci serviranno a capire meglio la struttura della Python Standard Library (da ora *PSL*) e come utilizzarla.

Per meglio comprendere le basi di I/O in Python costruiremo in seguito una classe *DirWalker* che data una directory principale visualizza sullo schermo o salva in un file, l'elenco di tutte le sottodirectory.

Basi di I/O

Le precedenti slide sull'organizzazione del codice ci serviranno a capire meglio la struttura della Python Standard Library (da ora *PSL*) e come utilizzarla.

Per meglio comprendere le basi di I/O in Python costruiremo in seguito una classe *DirWalker* che data una directory principale visualizza sullo schermo o salva in un file, l'elenco di tutte le sottodirectory.

Per farlo ci serviremo di alcune funzionalità di due moduli fondamentali della *PSL*: il modulo *os* ed il modulo *sys*.

Basi di I/O

Le precedenti slide sull'organizzazione del codice ci serviranno a capire meglio la struttura della Python Standard Library (da ora *PSL*) e come utilizzarla.

Per meglio comprendere le basi di I/O in Python costruiremo in seguito una classe *DirWalker* che data una directory principale visualizza sullo schermo o salva in un file, l'elenco di tutte le sottodirectory.

Per farlo ci serviremo di alcune funzionalità di due moduli fondamentali della *PSL*: il modulo *os* ed il modulo *sys*.

Nota: il codice che segue può essere semplificato utilizzando altre funzioni della *PSL*, ma per scopi didattici sono stati utilizzati meno *automatismi*.

dirwalker.py:

```
1  import os
2  import sys
3
4  class DirWalker(object):
5      def __init__(self, output = None):
6          self._check_output(output)
```

Importiamo innanzitutto i due moduli da utilizzare.

dirwalker.py:

```
1  import os
2  import sys
3
4  class DirWalker(object):
5      def __init__(self, output = None):
6          self._check_output(output)
```

Importiamo innanzitutto i due moduli da utilizzare.

E creiamo la nuova classe *DirWalker*.
Il costruttore accetta un parametro (opzionale) *output* e si aspetta che sia un file nel quale poter scrivere.
Questo parametro viene passato al metodo *check_output* che fa alcuni controlli.

```
8       def _check_output(self, output = None):
9           if output:
10              self.output = output
11          else:
12              self.output = sys.stdout
```

Quindi *check_output* se il parametro *output* non è nullo lo assegna all'attributo *output* dell'istanza, se invece il parametro è nullo viene assegnato *sys.stdout* ovvero l'output standard.

```
8       def _check_output(self, output = None):
9           if output:
10              self.output = output
11          else:
12              self.output = sys.stdout
```

Quindi *check_output* se il parametro *output* non è nullo lo assegna all'attributo *output* dell'istanza, se invece il parametro è nullo viene assegnato *sys.stdout* ovvero l'output standard.

Questo serve a fare in modo che se alla creazione di una DirWalker viene passato anche un file allora l'elenco delle direcotry sarà scritto su esso, altrimenti l'output andrà su schermo.

Segue poi la definizione del metodo *go*, il cuore della classe, che effettua un ciclo ricorsivo partendo dalla directory principale (passato come primo parametro) e stampa l'elenco delle sole directory.

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

Vediamo più in dettaglio come funziona questo metodo...

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

DirWalker stamperà le directory con percorso assoluto, quindi bisogna assicurarsi che il percorso dal quale si parte sia assoluto.
In caso contrario bisogna transformalo il percorso assoluto.
Questo è il compito delle righe 15 e 16.

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

La riga 15 controlla se il percorso passato *maindir* non è assoluto servendosi della funzione *isabs* del modulo *os.path*.

Nel caso in cui il percorso non sia assoluto, esso viene convertito tramite la funzione *os.path.abspath* che aggiunge il percorso della dir attuale a quello passato come parametro (riga 16).

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

Inizia a questo punto il ciclo for che analizza il contenuto della directory di partenza (riga 17).

Utilizziamo la funzione *os.listdir* che restituisce una lista di nomi di file e directory contenute nella directory passata come parametro.

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

Quindi per ogni elemento contenuto nel path *maindir* si inizia il ciclo di analisi.

La prima operazione è quella di creare il percorso completo del file o directory che si sta analizzando.
La variabile *maindir* contiene il percorso in cui cercare e *content* contiene il nome del file o directory che si sta analizzando.

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

Bisogna quindi legare le due variabili per ottenere il percorso completo.

Ci viene in aiuto la funzione *os.path.join* che unisce i due percorsi utilizzando il separatore di directory del sistema operativo sul quale è in esecuzione, *fondamentale allo sviluppo cross-platform* (riga 18).

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

Controlliamo poi se il percorso ottenuto in *path* è una directory servendoci della funzione *os.path.isdir* (riga 19).

In caso positivo scriviamo su *output* il percorso seguito dal ritorno a capo.
Il metodo *write* serve a scrivere dati su file.

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

Anche la classe *sys.stdout* possiede questo metodo, possiamo quindi utilizzarlo indistintamente se stiamo scrivendo su file o su schermo.

```
14      def go(self, maindir):
15          if not os.path.isabs(maindir):
16              maindir = os.path.abspath(maindir)
17          for content in os.listdir(maindir):
18              path = os.path.join(maindir, content)
19              if os.path.isdir(path):
20                  self.output.write(path + "\n")
21                  self.go(path)
```

Infine richiamiamo il metodo *go* sul nuovo percorso *path* per controllare se esistono altre directory all'interno e quindi ripetere il procedimento (riga 21).

Basi di I/O

A questo punto la nostra classe DirWalker è completa, ma per far funzionare il nostro programmino abbiamo bisogno che vengano letti i parametri in input e passati correttamente a DirWalker.

Basi di I/O

A questo punto la nostra classe DirWalker è completa, ma per far funzionare il nostro programmino abbiamo bisogno che vengano letti i parametri in input e passati correttamente a DirWalker.

Implementiamo di seguito il codice che permette al nostro modulo di ricevere due parametri in input: il primo è il percorso della directory da analizzare, il secondo (opzionale) è il nome del file in cui scrivere il risultato.

Basi di I/O

A questo punto la nostra classe DirWalker è completa, ma per far funzionare il nostro programmino abbiamo bisogno che vengano letti i parametri in input e passati correttamente a DirWalker.

Implementiamo di seguito il codice che permette al nostro modulo di ricevere due parametri in input: il primo è il percorso della directory da analizzare, il secondo (opzionale) è il nome del file in cui scrivere il risultato.

Vediamo di seguito come leggere parametri dalla riga di comando ed utilizzarli per il nostro scopo.

dirwalker.py:

```
24 if __name__ == "__main__":
25     path = sys.argv[1]
26
27     if len(sys.argv) >= 3:
28         f = open(sys.argv[2], "w")
29     else:
30         f = None
```

Il codice del modulo *dirwalker.py* continua con queste (ed altre) righe di codice.

Sicuramente la riga 24 colpisce maggiormente, essa è fondamentale al corretto funzionamento del modulo.

Basi di I/O

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

La variabile *__name__* contiene il nome del modulo attuale.
Se però il modulo è il principale, ovvero è il file da eseguire passato a Python nella riga di comando, allora questa variabile conterrà *__main__*.

Quindi, quale sarà il risultato?

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

Quando il modulo sarà chiamato direttamente sarà eseguito anche tutto il codice sotto l'if della riga 24.

Nel caso in cui il modulo sia stato importato da un altro modulo allora il codice dell'if non verrà eseguito.

Ciò è necessario affinchè il modulo posso essere importato correttamente da altri moduli.

Basi di I/O

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

Passiamo ora a leggere il primo parametro della riga di comando.

Utilizziamo la lista *sys.argv* che contiene tutti i parametri passati nella riga di comando di cui il primo elemento è sempre il nome del file principale: in questo caso *dirwalker.py*.

Basi di I/O

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

Quindi *sys.argv[1]* sarà il primo parametro passato e quindi il nostro percorso da analizzare. Lo salviamo nella variabile *path*.

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

Quindi *sys.argv[1]* sarà il primo parametro passato e quindi il nostro percorso da analizzare. Lo salviamo nella variabile *path*.

Siccome il secondo parametro (il file in cui scrivere) può essere opzionale, controlliamo prima che il numero di parametri passati nella riga di comando sia maggiore o uguale di 3.

Basi di I/O

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

In caso affermativo creiamo un nuovo file servendoci della funzione *open* di Python.

Basi di I/O

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

In caso affermativo creiamo un nuovo file servendoci della funzione *open* di Python.

I due parametri sono: il nome del file, in questo caso il secondo parametro della nostra linea di comando e, la modalità di apertura del file.

```
24  if __name__ == "__main__":
25      path = sys.argv[1]
26
27      if len(sys.argv) >= 3:
28          f = open(sys.argv[2], "w")
29      else:
30          f = None
```

In caso affermativo creiamo un nuovo file servendoci della funzione *open* di Python.

I due parametri sono: il nome del file, in questo caso il secondo parametro della nostra linea di comando e, la modalità di apertura del file.

Nel caso in cui il parametro non è presente inizializziamo *f* ad un valore nullo (riga 30).

Basi di I/O

```
32      dw = DirWalker(f)
33      dw.go(path)
34
35      if f:
36          f.close()
```

Il codice del modulo si conclude con queste 4 righe.

Basi di I/O

```
32      dw = DirWalker(f)
33      dw.go(path)
34
35      if f:
36          f.close()
```

Il codice del modulo si conclude con queste 4 righe.

Istanziamo un nuovo *DirWalker*, passandogli l'oggetto file su cui vogliamo scrivere il risultato (riga 32).

```
32      dw = DirWalker(f)
33      dw.go(path)
34
35      if f:
36          f.close()
```

Il codice del modulo si conclude con queste 4 righe.

Istanziamo un nuovo *DirWalker*, passandogli l'oggetto file su cui vogliamo scrivere il risultato (riga 32).

Come visto nella definizione della classe, quando viene passato un valore nullo l'output andrà su schermo.

```
32      dw = DirWalker(f)
33      dw.go(path)
34
35      if f:
36          f.close()
```

Il codice del modulo si conclude con queste 4 righe.

Istanziamo un nuovo *DirWalker*, passandogli l'oggetto file su cui vogliamo scrivere il risultato (riga 32).

Come visto nella definizione della classe, quando viene passato un valore nullo l'output andrà su schermo.

Avviamo quindi tutto il processo descritto prima chiamando il metodo *go* e passandogli il percorso ottenuto dalla riga di comando.

```
32      dw = DirWalker(f)
33      dw.go(path)
34
35      if f:
36          f.close()
```

Infine, se abbiamo aperto un file è buona norma chiuderlo per assicurarsi che tutte le operazioni di scrittura incomplete siano terminate (righe 35 e 36).

Il nostro script è ora pronto per essere avviato.
Possiamo ora chiamarlo da riga di comando, ad esempio:

```
# python dirwalker.py /home/utente
```

Esso elencherà su schermo tutte le directory presenti in */home/utente*.

Basi di I/O

Se invece aggiungiamo anche il parametro con il nome del file:

```
# python dirwalker.py /home/utente lista.txt
```

Troveremo nella stessa directory di esecuzione un file *lista.txt* contenente l'output che nell'esempio precedente è stato scritto su schermo.

Se invece aggiungiamo anche il parametro con il nome del file:

```
# python dirwalker.py /home/utente lista.txt
```

Troveremo nella stessa directory di esecuzione un file *lista.txt* contenente l'output che nell'esempio precedente è stato scritto su schermo.

Il nostro programmino è ora completo.

Se invece aggiungiamo anche il parametro con il nome del file:

```
# python dirwalker.py /home/utente lista.txt
```

Troveremo nella stessa directory di esecuzione un file *lista.txt* contenente l'output che nell'esempio precedente è stato scritto su schermo.

Il nostro programmino è ora completo.

Possiamo ritenerci soddisfatti?

Se invece aggiungiamo anche il parametro con il nome del file:

```
# python dirwalker.py /home/utente lista.txt
```

Troveremo nella stessa directory di esecuzione un file *lista.txt* contenente l'output che nell'esempio precedente è stato scritto su schermo.

Il nostro programmino è ora completo.

Possiamo ritenerci soddisfatti?

Non ancora! ;)

Gestione delle eccezioni

Cosa succede se la directory che indichiamo come sorgente non esiste?

Gestione delle eccezioni

Cosa succede se la directory che indichiamo come sorgente non esiste?

Cosa succede se non abbiamo il permesso di leggere una delle sottodirectory che il programma troverà nella sua elaborazione?

Gestione delle eccezioni

Cosa succede se la directory che indichiamo come sorgente non esiste?

Cosa succede se non abbiamo il permesso di leggere una delle sottodirectory che il programma troverà nella sua elaborazione?

E se il file dell'output non può essere scritto?

Gestione delle eccezioni

Cosa succede se la directory che indichiamo come sorgente non esiste?

Cosa succede se non abbiamo il permesso di leggere una delle sottodirectory che il programma troverà nella sua elaborazione?

E se il file dell'output non può essere scritto?

Durante la scrittura del file di output finisce lo spazio sul disco, come si comporterà il programma?

Gestione delle eccezioni

Cosa succede se la directory che indichiamo come sorgente non esiste?

Cosa succede se non abbiamo il permesso di leggere una delle sottodirectory che il programma troverà nella sua elaborazione?

E se il file dell'output non può essere scritto?

Durante la scrittura del file di output finisce lo spazio sul disco, come si comporterà il programma?

Vediamo come gestire le cosidette *eccezioni*!

Gestione delle eccezioni

Un eccezione è lo stato in cui un istruzione non può continuare la sua esecuzione a causa di un evento, di un errore insomma che non permette la corretta esecuzione del resto del codice.

Gestione delle eccezioni

Un eccezione è lo stato in cui un istruzione non può continuare la sua esecuzione a causa di un evento, di un errore insomma che non permette la corretta esecuzione del resto del codice.

Per *gestione delle eccezioni* si intende quindi quel meccanismo progettato per gestire questi errori permettendo di effettuare delle azioni in base al tipo di errore che si presenta.

Gestione delle eccezioni

Un eccezione è lo stato in cui un istruzione non può continuare la sua esecuzione a causa di un evento, di un errore insomma che non permette la corretta esecuzione del resto del codice.

Per *gestione delle eccezioni* si intende quindi quel meccanismo progettato per gestire questi errori permettendo di effettuare delle azioni in base al tipo di errore che si presenta.

Il concetto di eccezione non riguarda solo Python, è tipico di molti linguaggi come Java, C++, Perl, Smalltalk, Javascript, Objective-C, ecc.

Gestione delle eccezioni

Se proviamo ad eseguire il seguente codice:

```
1  f = open("/fileNonEsistente", "r")
2  print f.readline()
```

potremmo imbatterci in vari errori come ad esempio nel caso in cui il file che vogliamo aprire non esiste oppure non abbiamo i permessi sufficienti ad aprirlo.

Gestione delle eccezioni

Se proviamo ad eseguire il seguente codice:

```
1  f = open("/fileNonEsistente", "r")
2  print f.readline()
```

potremmo imbatterci in vari errori come ad esempio nel caso in cui il file che vogliamo aprire non esiste oppure non abbiamo i permessi sufficienti ad aprirlo.

In questo caso la funzione *open* emetterà un eccezione di tipo `IOError` e il programma terminerà visualizzando alcune informazioni sull'errore avvenuto.

Gestione delle eccezioni

Se proviamo ad eseguire il seguente codice:

```
1  f = open("/fileNonEsistente", "r")
2  print f.readline()
```

potremmo imbatterci in vari errori come ad esempio nel caso in cui il file che vogliamo aprire non esiste oppure non abbiamo i permessi sufficienti ad aprirlo.

In questo caso la funzione *open* emetterà un eccezione di tipo `IOError` e il programma terminerà visualizzando alcune informazioni sull'errore avvenuto.

Se non vogliamo che il programma si chiuda dobbiamo catturare l'eccezione e gestirla al meglio.

Gestione delle eccezioni

Ecco un semplice esempio di gestione dell'eccezione *IOError* con conseguente semplice azione:

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Gestione delle eccezioni

Ecco un semplice esempio di gestione dell'eccezione *IOError* con conseguente semplice azione:

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Il comando `try` indica a Python che si intende gestire dei tipi di eccezioni sul blocco di codice contenuto al suo interno.

Gestione delle eccezioni

Ecco un semplice esempio di gestione dell'eccezione *IOError* con conseguente semplice azione:

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Il comando `try` indica a Python che si intende gestire dei tipi di eccezioni sul blocco di codice contenuto al suo interno.

Con il comando `except` si indica invece quali tipi di errori si vuole gestire.

Gestione delle eccezioni

Ecco un semplice esempio di gestione dell'eccezione *IOError* con conseguente semplice azione:

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Il comando `try` indica a Python che si intende gestire dei tipi di eccezioni sul blocco di codice contenuto al suo interno.

Con il comando `except` si indica invece quali tipi di errori si vuole gestire.

Vediamo subito il funzionamento in esecuzione.

Gestione delle eccezioni

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Se il file non esiste l'istruzione di riga 2 emetterà un eccezione di tipo *IOError* ed il resto del codice contenuto nel blocco *try* non sarà eseguito.

Gestione delle eccezioni

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Se il file non esiste l'istruzione di riga 2 emetterà un eccezione di tipo *IOError* ed il resto del codice contenuto nel blocco *try* non sarà eseguito.

Python cercherà un blocco *except* che gestisca l'eccezione *IOError*.

Gestione delle eccezioni

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Se il file non esiste l'istruzione di riga 2 emetterà un eccezione di tipo *IOError* ed il resto del codice contenuto nel blocco *try* non sarà eseguito.

Python cercherà un blocco *except* che gestisca l'eccezione *IOError*.

Se lo trova esegue il codice che trova all'interno e continua l'esecuzione del programma.
In questo caso stamperà a schermo il messaggio dell'errore che l'eccezione ha portato con se.

Gestione delle eccezioni

Vediamo meglio la riga 4, ovvero quella responsabile della gestione dell' eccezione.

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Gestione delle eccezioni

Vediamo meglio la riga 4, ovvero quella responsabile della gestione dell' eccezione.

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

La sintassi è uguale per tutti i tipi di eccezioni ed è molto semplice.

Gestione delle eccezioni

Vediamo meglio la riga 4, ovvero quella responsabile della gestione dell' eccezione.

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

La sintassi è uguale per tutti i tipi di eccezioni ed è molto semplice.

Il comando *except* deve essere seguito dal tipo di eccezione che si vuole catturare e opzionalmente dal nome della variabile che conterrà l'istanza dell'eccezione catturata.

Gestione delle eccezioni

Vediamo meglio la riga 4, ovvero quella responsabile della gestione dell' eccezione.

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

La sintassi è uguale per tutti i tipi di eccezioni ed è molto semplice.

Il comando *except* deve essere seguito dal tipo di eccezione che si vuole catturare e opzionalmente dal nome della variabile che conterrà l'istanza dell'eccezione catturata.

Nel nostro caso le informazioni sull'eccezione saranno contenute dalla variabile *ex*.

Gestione delle eccezioni

Ogni eccezione ha i suoi attributi che servono a dare al programma informazioni dettagliate sull'errore avvenuto.

Gestione delle eccezioni

Ogni eccezione ha i suoi attributi che servono a dare al programma informazioni dettagliate sull'errore avvenuto.

Nel nostro esempio utilizziamo solo l'attributo *message* (contenuto in tutti tipi di eccezioni) per stampare un messagio di descrizione dell'errore su schermo.

Gestione delle eccezioni

Ogni eccezione ha i suoi attributi che servono a dare al programma informazioni dettagliate sull'errore avvenuto.

Nel nostro esempio utilizziamo solo l'attributo *message* (contenuto in tutti tipi di eccezioni) per stampare un messagio di descrizione dell'errore su schermo.

Come abbiamo visto il programma scrive un messaggio ed esce, niente di speciale, ma lo stesso procedimento è usato in situazioni più complesse per eseguire codice che gestisca l'errore e continui l'esecuzione del programma.

Gestione delle eccezioni

Ogni eccezione ha i suoi attributi che servono a dare al programma informazioni dettagliate sull'errore avvenuto.

Nel nostro esempio utilizziamo solo l'attributo *message* (contenuto in tutti tipi di eccezioni) per stampare un messagio di descrizione dell'errore su schermo.

Come abbiamo visto il programma scrive un messaggio ed esce, niente di speciale, ma lo stesso procedimento è usato in situazioni più complesse per eseguire codice che gestisca l'errore e continui l'esecuzione del programma.

Il nostro codice è ora più sicuro perchè gestisce l'errore di accesso al file.

Gestione delle eccezioni

Ma che succede se il file è vuoto?

Gestione delle eccezioni

Ma che succede se il file è vuoto?

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Il metodo *readline* dell'oggetto *file* legge una riga dal file, avanza il puntatore alla riga successiva e restituisce la riga letta.

Gestione delle eccezioni

Ma che succede se il file è vuoto?

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
```

Il metodo *readline* dell'oggetto *file* legge una riga dal file, avanza il puntatore alla riga successiva e restituisce la riga letta.

Se il file è vuoto o è finito sarà emessa un eccezione di tipo *EOFError* (End Of File Error), che indica l'impossibilità di leggere dati dal file e quindi l'impossibilità di completare l'esecuzione di *readline*.

Gestione delle eccezioni

Siccome ancora non abbiamo gestito l'eccezione di tipo *EOFError* il programma terminerà segnalando l'errore.

Aggiungiamo al nostro codice anche la gestione di questo tipo di errore nel modo seguente:

```
1  try:
2      f = open("/fileNonEsistente", "r")
3      print f.readline()
4  except IOError, ex:
5      print "Errore " + ex.message
6  except EOFError:
7      print "Il file e' vuoto o si e' arrivati alla fine!"
```

Nelle righe 6 e 7 definiamo la cattura dell'eccezione e le istruzioni da eseguire.

Gestione delle eccezioni

Possiamo ora essere abbastanza soddisfatti della gestione degli errori aggiunta al nostro codice.

Gestione delle eccezioni

Possiamo ora essere abbastanza soddisfatti della gestione degli errori aggiunta al nostro codice.

Ora che conosciamo meglio il significato di gestire eccezioni, vediamo più in dettaglio cosa sono precisamente le eccezioni, come sono organizzate e come crearne di nuove.

Gestione delle eccezioni

Possiamo ora essere abbastanza soddisfatti della gestione degli errori aggiunta al nostro codice.

Ora che conosciamo meglio il significato di gestire eccezioni, vediamo più in dettaglio cosa sono precisamente le eccezioni, come sono organizzate e come crearne di nuove.

Le eccezioni sono semplici classi Python che ereditano da *Exception*, la classe in cima alla gerarchia delle eccezioni.

Gestione delle eccezioni

Possiamo ora essere abbastanza soddisfatti della gestione degli errori aggiunta al nostro codice.

Ora che conosciamo meglio il significato di gestire eccezioni, vediamo più in dettaglio cosa sono precisamente le eccezioni, come sono organizzate e come crearne di nuove.

Le eccezioni sono semplici classi Python che ereditano da *Exception*, la classe in cima alla gerarchia delle eccezioni.

Ogni eccezione in Python ed ogni nuova eccezione che vogliamo creare deve ereditare da essa le funzionalità base per essere riconosciuta da Python come eccezione.

Gestione delle eccezioni

Per capire meglio questi concetti applichiamo la gestione delle eccezioni al nostro modulo *dirwalker* facendo alcune riflessioni.

Gestione delle eccezioni

Per capire meglio questi concetti applichiamo la gestione delle eccezioni al nostro modulo *dirwalker* facendo alcune riflessioni.

Al momento *dirwalker* non gestisce alcuna eccezione lasciando quindi al modulo chiamante il compito di gestire le eccezioni in cui si può imbattere.

Gestione delle eccezioni

Per capire meglio questi concetti applichiamo la gestione delle eccezioni al nostro modulo *dirwalker* facendo alcune riflessioni.

Al momento *dirwalker* non gestisce alcuna eccezione lasciando quindi al modulo chiamante il compito di gestire le eccezioni in cui si può imbattere.

Ciò è in genere sbagliato perchè si presuppone che il modulo che chiama *dirwalker* conosca perfettamente cosa esso fa e quali problemi può causare.

Gestione delle eccezioni

Per capire meglio questi concetti applichiamo la gestione delle eccezioni al nostro modulo *dirwalker* facendo alcune riflessioni.

Al momento *dirwalker* non gestisce alcuna eccezione lasciando quindi al modulo chiamante il compito di gestire le eccezioni in cui si può imbattere.

Ciò è in genere sbagliato perchè si presuppone che il modulo che chiama *dirwalker* conosca perfettamente cosa esso fa e quali problemi può causare.

Per facilitare l'utilizzo di *dirwalker* abbiamo bisogno di focalizzare meglio quali sono i problemi e come semplificare la loro gestione.

Gestione delle eccezioni

Gli errori in cui il nostro *dirwalker* può imbattersi sono vari, ma possono essere semplificati focalizzando due problemi principali:

- La directory in cui cercare non esiste o non si hanno i permessi per accedervi.
- Non si ha il permesso di scrivere nel file specificato per l'output.

L'esigenza di dettagliare maggiormente gli errori dipende dall'applicazione.
Per la nostra applicazione possiamo ritenere soddisfacente questo tipo di semplificazione.

Gestione delle eccezioni

Adesso che abbiamo focalizzato meglio quali sono gli errori creiamo subito le nuove eccezioni per il modulo *dirwalker*.

Gestione delle eccezioni

Adesso che abbiamo focalizzato meglio quali sono gli errori creiamo subito le nuove eccezioni per il modulo *dirwalker*.

É buona norma creare innanzitutto un eccezione generica per il modulo. Da essa erediteranno tutte le altre eccezioni dello stesso modulo. Il perchè di questa scelta lo si comprenderà meglio in seguito.

```
1 class Error(Exception):
2     pass
```

Abbiamo così definito una nuova eccezione *dirwalker.Error* ereditando tutte le funzionalità dalla classe base *Exception* e senza definirne altre.

Gestione delle eccezioni

Creiamo ora l'eccezione da emettere quando la directory di input per *dirwalker* non è valida:

```
1  class InputDirError(Error):
2      def __init__(self, inputdir):
3          self.inputdir = inputdir
```

Gestione delle eccezioni

Creiamo ora l'eccezione da emettere quando la directory di input per *dirwalker* non è valida:

```
1  class InputDirError(Error):
2      def __init__(self, inputdir):
3          self.inputdir = inputdir
```

Viene definito anche il costruttore per questa eccezione per permettere di aggiungere un informazione da passare all'emissione, ovvero il percorso che ha causato il problema.

Gestione delle eccezioni

Creiamo ora l'eccezione da emettere quando la directory di input per *dirwalker* non è valida:

```
1 class InputDirError(Error):
2     def __init__(self, inputdir):
3         self.inputdir = inputdir
```

Viene definito anche il costruttore per questa eccezione per permettere di aggiungere un informazione da passare all'emissione, ovvero il percorso che ha causato il problema.

Deve ovviamente avere un nome che possa facilmente collegarla al tipo di errore per cui viene emessa.

Gestione delle eccezioni

Creiamo ora l'eccezione da emettere quando la directory di input per *dirwalker* non è valida:

```
1 class InputDirError(Error):
2     def __init__(self, inputdir):
3         self.inputdir = inputdir
```

Viene definito anche il costruttore per questa eccezione per permettere di aggiungere un informazione da passare all'emissione, ovvero il percorso che ha causato il problema.

Deve ovviamente avere un nome che possa facilmente collegarla al tipo di errore per cui viene emessa.

Come si vede eredita da *Error* l'eccezione definita prima.

Gestione delle eccezioni

Creiamo infine l'ultima eccezione da emettere quando si verificano problemi nella scrittura del file di output:

```
1 class OutputFileError(Error):
2     def __init__(self, outputfile):
3         self.outputfile = outputfile
```

Anche per questa eccezione abbiamo un informazione da salvare, ovvero il percorso del file che ha causato il problema di scrittura.

Gestione delle eccezioni

Vediamo ora come gestire le eccezioni del modulo *dirwalker* ed emettere le nuove eccezioni al momento appropriato. Modifichiamo innanzitutto il metodo *DirWalker.go* per gestire i due tipi di errori possibili:

```
14      def go(self, maindir):
15          # ...
16          try:
17              for content in os.listdir(maindir):
18                  path = os.path.join(maindir, content)
19                  if os.path.isdir(path):
20                      self.output.write(path + "\n")
21                      self.go(path)
22          except OSError:
23              raise InputDirError(maindir)
24          except IOError:
25              raise OutputFileError(self.output.name)
```

Gestione delle eccezioni

```
14      def go(self, maindir):
15          # ...
16          try:
17              for content in os.listdir(maindir):
18                  path = os.path.join(maindir, content)
19                  if os.path.isdir(path):
20                      self.output.write(path + "\n")
21                      self.go(path)
22          except OSError:
23              raise InputDirError(maindir)
24          except IOError:
25              raise OutputFileError(self.output.name)
```

Lasciamo il metodo pressocchè invariato, racchiudendo soltanto il ciclo for in un blocco *try* gestendo gli errori *OSError* e *IOError*.

Gestione delle eccezioni

```
14      def go(self, maindir):
15          # ...
16          try:
17              for content in os.listdir(maindir):
18                  path = os.path.join(maindir, content)
19                  if os.path.isdir(path):
20                      self.output.write(path + "\n")
21                      self.go(path)
22          except OSError:
23              raise InputDirError(maindir)
24          except IOError:
25              raise OutputFileError(self.output.name)
```

La funzione *os.listdir* emetterà un *OSError* nel caso in cui ci siano problemi a scorrere il contenuto di *maindir*.

Gestione delle eccezioni

```
14      def go(self, maindir):
15          # ...
16          try:
17              for content in os.listdir(maindir):
18                  path = os.path.join(maindir, content)
19                  if os.path.isdir(path):
20                      self.output.write(path + "\n")
21                      self.go(path)
22          except OSError:
23              raise InputDirError(maindir)
24          except IOError:
25              raise OutputFileError(self.output.name)
```

Catturato l'errore *OSError*, viene emesso con il comando `raise` un nuovo tipo di errore *InputDirError* che dovrà essere gestito dal codice chiamante.

Gestione delle eccezioni

```
14      def go(self, maindir):
15          # ...
16          try:
17              for content in os.listdir(maindir):
18                  path = os.path.join(maindir, content)
19                  if os.path.isdir(path):
20                      self.output.write(path + "\n")
21                      self.go(path)
22          except OSError:
23              raise InputDirError(maindir)
24          except IOError:
25              raise OutputFileError(self.output.name)
```

Stessa cosa per l'errore *IOError* che emette il metodo *write* se ci sono problemi nella scrittura del file.

Gestione delle eccezioni

```
14      def go(self, maindir):
15          # ...
16          try:
17              for content in os.listdir(maindir):
18                  path = os.path.join(maindir, content)
19                  if os.path.isdir(path):
20                      self.output.write(path + "\n")
21                      self.go(path)
22          except OSError:
23              raise InputDirError(maindir)
24          except IOError:
25              raise OutputFileError(self.output.name)
```

Viene catturato l'errore *IOError* ed emesso un nuovo tipo di errore *OutputFileError* che dovrà essere gestito nel codice chiamante.

Gestione delle eccezioni

Modifichiamo ora il codice principale del modulo per gestire i restanti errori.

```
42  if __name__ == "__main__":
43      path = sys.argv[1]
44
45      if len(sys.argv) >= 3:
46          try:
47              f = open(sys.argv[2], "w")
48          except IOError:
49              sys.exit("Non e' possibile scrivere nel \
50                        file %s" % sys.argv[2])
51      else:
52          f = None
```

É stato semplicemente gestito l'errore di apertura di un file in scrittura quando non si hanno i permessi.

Gestione delle eccezioni

Gestiamo infine nel codice principale i nuovi tipi di errore creati:

```
54      try:
55          dw = DirWalker(f)
56          dw.go(path)
57      except InputDirError, ex:
58          print "La directory di input %s \
59                  non e' valida" % ex.inputdir
60      except OutputFileError, ex:
61          print "Errore nella scrittura \
62                  del file %s" % ex.outputfile
```

Racchiudiamo le operazioni sensibili di *DirWalker* in un blocco *try* gestendo i nuovi tipi di errore e utilizzando gli attributi personalizzati (righe 59 e 62).

Gestione delle eccezioni

Se proviamo ora ad avviare il programma con varie combinazioni di parametri vedremo che non vengono più restituiti i messaggi di errore di Python, ma vengono stampati a video i messaggi gestiti nel codice principale.

Gestione delle eccezioni

Se proviamo ora ad avviare il programma con varie combinazioni di parametri vedremo che non vengono più restituiti i messaggi di errore di Python, ma vengono stampati a video i messaggi gestiti nel codice principale.

L'esempio appena visto è molto semplice, forse troppo per capire realmente l'utilità della gestione delle eccezioni.
É necessario esercizio per capire meglio quanto questi meccanismi possano aiutare a tenere sotto controllo gli errori e scrivere codice pulito.

Gestione delle eccezioni

Se proviamo ora ad avviare il programma con varie combinazioni di parametri vedremo che non vengono più restituiti i messaggi di errore di Python, ma vengono stampati a video i messaggi gestiti nel codice principale.

L'esempio appena visto è molto semplice, forse troppo per capire realmente l'utilità della gestione delle eccezioni. É necessario esercizio per capire meglio quanto questi meccanismi possano aiutare a tenere sotto controllo gli errori e scrivere codice pulito.

Chi avrà programmato in C ad esempio saprà che spesso la gestione degli errori si traduce in una serie di *if* annidati che spesso rendono difficile la lettura del codice e quindi la ricerca di bug. Potete dimenticarvi di questo tipo di problemi programmando in Python.

Gestione delle eccezioni

Concludiamo questa lezione dando altri piccoli chiarimenti sulla gestione delle eccezioni.
Ciò allo scopo di facilitare l'applicazione di questi concetti a programmi più complessi e per evitare i tipici errori.

Gestione delle eccezioni

Concludiamo questa lezione dando altri piccoli chiarimenti sulla gestione delle eccezioni.
Ciò allo scopo di facilitare l'applicazione di questi concetti a programmi più complessi e per evitare i tipici errori.

É bene chiarire come Python scelga quale codice eseguire a seconda dell' eccezione che viene emessa.
Assimilare ora questi concetti permetterà di capire dove si sta sbagliando quando sembrerà che Python non faccia ciò che gli diciamo.

Gestione delle eccezioni

Come visto in precedenza ogni eccezione deve ereditare da *Exception* il tipo di eccezione principale.

Gestione delle eccezioni

Come visto in precedenza ogni eccezione deve ereditare da *Exception* il tipo di eccezione principale.

Ciò crea una reale gerarchia di eccezioni in cui ognuna è figlia di una altra e può essere genitore di altre eccezioni.

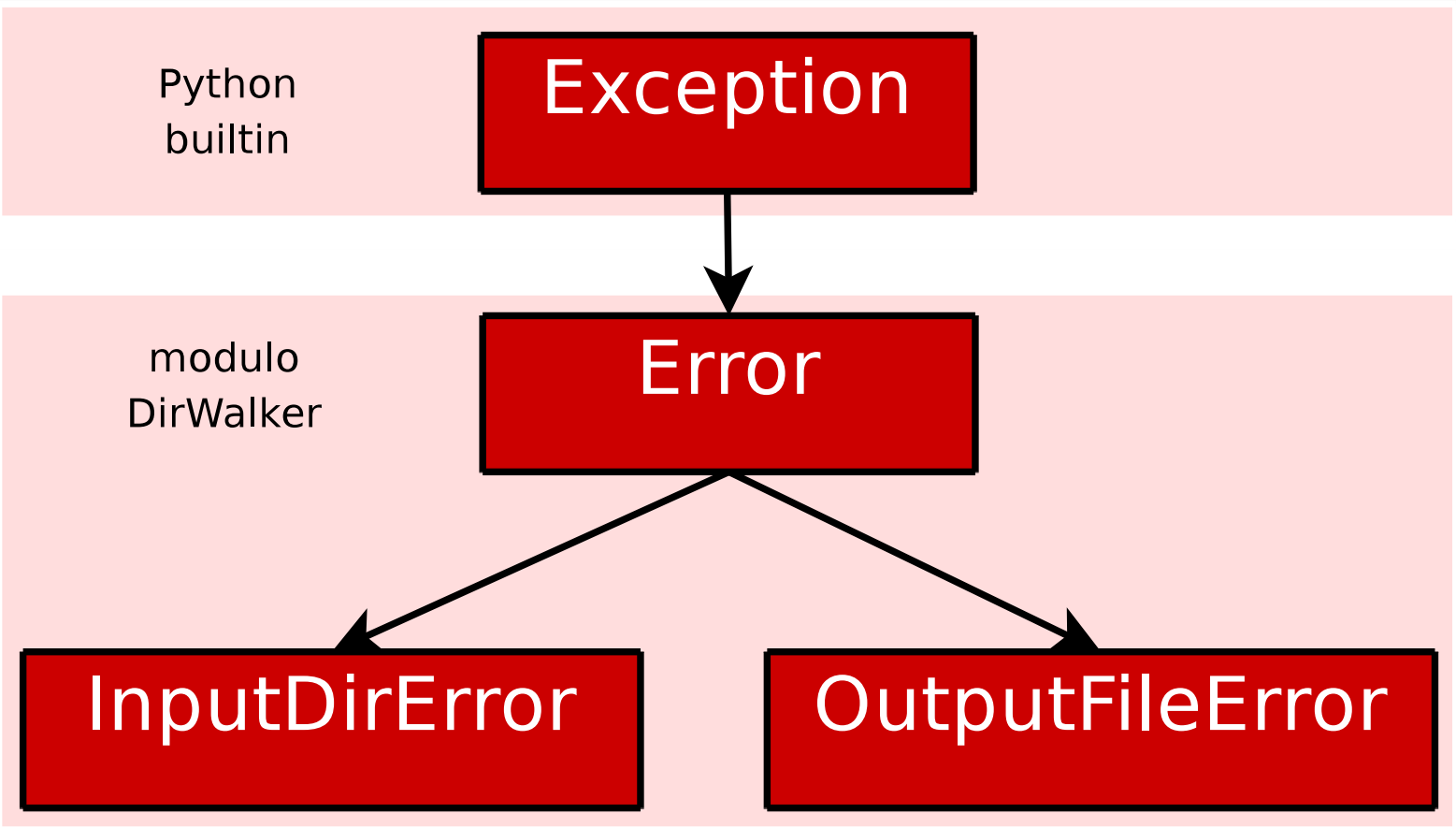
Gestione delle eccezioni

Come visto in precedenza ogni eccezione deve ereditare da *Exception* il tipo di eccezione principale.

Ciò crea una reale gerarchia di eccezioni in cui ognuna è figlia di una altra e può essere genitore di altre eccezioni.

Python utilizza questa gerarchia per decidere i blocchi di codice negli *except* da eseguire all'emissione di una eccezione.

Nel modulo *dirwalker* abbiamo definito tre nuove eccezioni creando la seguente gerarchia:



Si nota chiaramente che l'errore generico di *dirwalker* è figlio, ovviamente, di *exception* e che a sua volta è genitore delle eccezioni *InputDirError* e *OutputFileError*.

Gestione delle eccezioni

Il blocco di codice precedente per la gestione degli errori di *DirWalker*, di seguito:

```
54      try:
55          dw = DirWalker(f)
56          dw.go(path)
57      except InputDirError, ex:
58          print "La directory di input %s \
59                  non e' valida" % ex.inputdir
60      except OutputFileError, ex:
61          print "Errore nella scrittura \
62                  del file %s" % ex.outputfile
```

può essere modificato utilizzando un solo *except* per tutte le eccezioni definite nel modulo *dirwalker*.

Gestione delle eccezioni

Modifichiamo il codice in questo modo:

```
54      try:
55          dw = DirWalker(f)
56          dw.go(path)
57      except Error:
58          print "Errore DirWalker!"
```

Questa modifica, anche se non utile al funzionamento del programma, ci serve a capire la logica che guida la gestione delle eccezioni.

In questo caso il codice del blocco *except* verrà eseguito sia che l'eccezione sia *InputDirError* che se sia *OutputFileError*, perchè entrambe le eccezioni sono istanze del loro genitore *Error*.

Gestione delle eccezioni

Importantissimo è anche l'ordine dei blocchi *except* perchè essi vengono analizzati uno per volta dal primo all'ultimo e solo uno di essi riceverà la gestione dell'errore.
Quindi in questo caso:

```
54      try:
55          dw = DirWalker(f)
56          dw.go(path)
57      except InputDirError, ex:
58          print "La directory di input %s non e' valida" % ex.input
59      except Error:
60          print "Errore DirWalker!"
```

se l'eccezione emessa è *InputDirError* verrà gestita dal primo blocco, altrimenti nel secondo blocco saranno gestite le altre eccezioni del modulo *dirwalker* (solo *OutputFileError* in questo caso).

Gestione delle eccezioni

Invertendo l'ordine dei blocchi *except* il risultato cambia:

```
54      try:
55          dw = DirWalker(f)
56          dw.go(path)
57      except Error:
58          print "Errore DirWalker!"
59      except InputDirError, ex:
60          print "La directory di input %s non e' valida" % ex.inpu
```

se viene emesso un *InputDirError* verrà gestito dal primo blocco inquanto è istanza del genitore *Error*, con la conseguenza che il secondo blocco non verrà mai eseguito.

Fate particolare attenzione all'organizzazione delle eccezioni!

Gestione delle eccezioni

Concludiamo questa lezioni con l'ultimo blocco di gestione eccezioni: il blocco *finally*.

Il codice scritto nel blocco *finally* viene eseguito sempre, sia che il codice emetta un eccezione sia che venga eseguito tutto correttamente.
Ad esempio:

```
1 try:
2     f = open("miofile", "w")
3     f.writeline("scrivo nel mio file")
4 except IOError:
5     print "Errore nel mio file"
6     sys.exit(-1)
7 finally:
8     f.close()
```

Gestione delle eccezioni

Se dovesse presentarsi un problema nella scrittura del file l'errore sarebbe gestito stampando un messaggio e chiudendo il programma (riga 6).

Gestione delle eccezioni

Se dovesse presentarsi un problema nella scrittura del file l'errore sarebbe gestito stampando un messaggio e chiudendo il programma (riga 6).

Prima di lasciare il blocco try Python eseguo tutto il codice presente nel blocco finally. Ciò per assicurare la corretta finalizzazione delle operazioni anche in caso di errore.

Gestione delle eccezioni

Se dovesse presentarsi un problema nella scrittura del file l'errore sarebbe gestito stampando un messaggio e chiudendo il programma (riga 6).

Prima di lasciare il blocco try Python eseguo tutto il codice presente nel blocco finally. Ciò per assicurare la corretta finalizzazione delle operazioni anche in caso di errore.

Ma `f.close()` non poteva essere inserito nel blocco *except* ottenendo lo stesso risultato?

Gestione delle eccezioni

Se dovesse presentarsi un problema nella scrittura del file l'errore sarebbe gestito stampando un messaggio e chiudendo il programma (riga 6).

Prima di lasciare il blocco try Python eseguo tutto il codice presente nel blocco finally. Ciò per assicurare la corretta finalizzazione delle operazioni anche in caso di errore.

Ma `f.close()` non poteva essere inserito nel blocco *except* ottenendo lo stesso risultato?

In questo caso si, ma attenzione!
Di blocco *finally* ce ne può essere soltanto uno e deve contenere il codice di finalizzazione comune a tutte le eccezioni che vengono gestite.

In questo modo si evita di riscrivere più volte il codice di finalizzazione.

Conclusioni

Finisce così questa terza lezione, in cui abbiamo analizzato molti concetti fondamentali alla realizzazione di programmi più ampi degli esempi visti.

Per fissare meglio questi argomenti prova a modificare il codice del modulo *dirwalker* aggiungendo nuove funzionalità o analizzando altri probabili errori che si possono presentare.

Che altro si può aggiungere?
Implementiamo la scelta di considerare o meno le directory nascoste.
DirWalker non continua la scansione della directory se trova un problema nel percorso. Prova ad implementare un meccanismo che ignori questi errori e continui la scansione.

Alla prossima lezione!