Pattern Recognition
and Applications Lab

La Programmazione ad Oggetti in Python

Docente: Ambra Demontis

Anno Accademico: 2020 - 2021

University of Cagliari,
Italy

Department of Electrical and
Electronic Engineering

Corso di Laurea in Ingegneria Elettrica, Elettronica e Informatica

La Programmazione ad Oggetti in Python

In queste slide vedremo come:

- Sollevare eccezioni
- Gestire eccezioni
- Definire nuove eccezioni

Le Eccezioni

Durante l’esecuzione di un programma può capitare che alcuni input dell’utente o risultati di calcoli siano **invalidi** o **imprevisti**.

Ad esempio, dei risultati ottenuti possono portare alla divisione di un numero per zero.

Questi eventi generano spesso delle eccezioni.

Le Eccezioni

Alcuni esempi di eccezioni sono:

```
x = 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
        x = 5 / 0
ZeroDivisionError: division by zero

lst = ["a", "b", "c"]
print(lst[3])
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
        print(lst[3])
IndexError: list index out of range
```

Le Eccezioni

Alcuni esempi di eccezioni sono:

```
a = [3] + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
        a = [3] + 3
TypeError: can only concatenate list (not "int") to list
```

Le Eccezioni

Le eccezioni:

1. Comunicano all'utente quale riga di codice ha generato il problema.
2. Comunicano qual'è il problema.
3. Terminano l'esecuzione del programma.

Esempio:

```
x = 5 / 0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    x = 5 / 0
ZeroDivisionError: division by zero
```

1

2

Sollevare Eccezioni

Le eccezioni **possono essere sollevate dal programmatore per informare l'utente che è avvenuto un evento imprevisto**.

La sintassi per sollevare un'eccezione è:
raise <tipo_eccezione> ("messaggio per l'utente")

Sollevare Eccezioni

In Python esistono diversi tipi di eccezioni. Quelli più comunemente utilizzati dai programmatori sono:

ValueError: un'operazione o funzione riceve un argomento che è di un tipo previsto ma ha un valore inappropriato.

TypeError: un'operazione o funzione è applicata ad un oggetto di tipo inappropriato.

NotImplementedError: indica che il valore ricevuto da un'operazione o da una funzione non è contemplato dall'attuale implementazione.

Sollevare Eccezioni

Consideriamo la classe CEsame implementata come mostrato sotto:

```
class CEsame:
  def __init__(self, nome, voto):
    self._nome = nome
    self._voto = voto

  @property
  def nome(self):
    return self._nome

  @property
  def voto(self):
    return self._voto
```

Sollevare Eccezioni

Supponiamo di voler far si che venga generata un'eccezione se:

1) Il voto non è compreso tra 0 e 30 (il nostro programma prevede voti in questo range)
2) Il voto non è un intero

Controllo di Tipo

Per controllare se un oggetto appartiene esattamente al tipo desiderato possiamo utilizzare la funzione **type** e l'operatore **is** in questo modo:

type(<oggetto>) is <tipo_desiderato>

Esempio:

```
if type(3) is int:
  print("tipo intero")
else:
  print("altro tipo")
```

Stamperà: tipo intero

Controllo di Tipo

```
if type(3.5) is float:
  print("tipo frazionario")
else:
  print("altro tipo")
```

Stamperà: tipo frazionario

```
if type("LPO") is str:
  print("tipo stringa")
else:
  print("altro tipo")
```

Stamperà: tipo stringa

Sollevare Eccezioni

Il valore dell'attributo che vogliamo controllare viene memorizzato nel metodo __init__

```
class CEsame:

  def __init__(self, nome, voto):
     self._nome = nome
     self._voto = voto

     …
```

Quindi è sufficiente modificare questo metodo.

Sollevare Eccezioni

Possiamo far si che questo metodo richiami una funzione che si occupa di effettuare i controlli sul valore di voto ed eventualmente sollevare un'eccezione.

```
class CEsame:

  def __init__(self, nome, voto):

     self._controlla_valore_voto(voto)

     self._nome = nome
     self._voto = voto
    ...
```

Sollevare Eccezioni

```
def _controlla_valore_voto(self, voto):

    if type(voto) is not int:
        raise TypeError("Il valore dell'attributo voto deve essere un "
                        "intero")

    if voto < 0 or voto > 30:
        raise ValueError("Il valore dell'attributo voto deve essere "
                         "compreso tra zero e 30")
```

Sollevare Eccezioni

Avendo modificato il codice come mostrato verrà sollevata un'eccezione nel caso in cui il valore dell'attributo voto passato come argomento al metodo __init__ non sia conforme a quanto il programmatore si aspetta.

Esempio:

```
oggetto_esame = CEsame('LPO', "50")

Traceback (most recent call last):
  ...
TypeError: Il valore dell'attributo voto deve essere un intero
```

Perchè Sollevare Eccezioni?

Non potremmo semplicemente inserire una stampa di errore?

```
def _controlla_valore_voto(self, voto):

    if type(voto) is not int:
      print("Il valore dell'attributo voto deve essere un intero")

    if voto < 0 or voto > 30:
       print("Il valore dell'attributo voto deve essere compreso tra zero e 30")
```

NO, non si può fare!

1) Il programma continuerebbe generando risultati errati.
2) La stampa di errore potrebbe non essere vista dall'utente che utilizzerebbe i risultati errati generati dal programma.

Perchè Sollevare Eccezioni?

- Permette di **terminare subito il programma, liberando le risorse e segnalando** all'utente **che è avvenuto un evento imprevisto** evitando che questo continui con valori invalidi generando dei risultati errati*.

- Permette di fornire all'utente dei messaggi di errore ad-hoc e più informativi di quelli che potrebbero venire generati altrimenti.

* Sempre più spesso i calcoli vengono effettuati su costose GPU. Tenerle occupate inutilmente con un programma che non andrà a buon fine costituisce uno spreco di risorse.

Esercizio: Sollevare Eccezioni

Creare una classe chiamata Prodotto con due attributi privati chiamati *idx_prodotto* e *prezzo* acquisiti al momento dell'inizializzazione dell'oggetto.

Il metodo inizializzatore deve controllare che il valore dell'attributo prezzo ricevuto sia maggiore di zero e in caso contrario sollevare un'eccezione.

Esercizio: Sollevare Eccezioni

```
class CProdotto:

  def _controlla_prezzo(self, value):

    if value < 0:
      raise ValueError("Il prezzo deve essere maggiore di zero ")

  def __init__(self, idx_prodotto, prezzo):

    self._controlla_prezzo(prezzo)
    self._idx_prodotto = idx_prodotto
    self._prezzo = prezzo
```

Esercizio: Sollevare Eccezioni

```
oggetto_prodotto = CProdotto("a56", 4)
print("primo oggetto creato")
oggetto_prodotto = CProdotto("a56", -4)
print("secondo oggetto creato")
```

Gestire Eccezioni

Le eccezioni possono essere gestite dal programmatore.

Scrivendo del codice ad-hoc può intercettarle, evitando che il programma si blocchi ed effettuare delle operazioni per risolvere i problemi da esse segnalati.

Gestire Eccezioni

Per gestire un'eccezione si utilizza la seguente sintassi:

try:

… (codice da eseguire se l'eccezione non si verifica)

except <tipo_eccezione>:

… (codice da eseguire se un'eccezione di tipo <tipo_eccezione> si verifica)

Gestire Eccezioni

Nel caso in cui si vogliano gestire n tipi di eccezioni differenti utilizzando lo stesso codice:

try:
    … (codice da eseguire se l'eccezione non si verifica)

except (<tipo_eccezione_1>, .. <tipo_eccezione_n>):
    … (codice da eseguire se un'eccezione di uno dei tipi specificati si verifica)

Gestire Eccezioni

Nel caso in cui si voglia gestire con lo stesso codice tutte le eccezioni causate da un evento imprevisto generato dal codice si utilizza la sintassi:

```
try:
        … (codice da eseguire se l'eccezione non si verifica)

except Exception:
        … (codice da eseguire se un'eccezione di un tipo qualsiasi si verifica )
```

Gestire Eccezioni

Nel caso in cui si vogliano gestire tipi di eccezioni differenti svolgendo nel caso in cui essi si verifichino operazioni differenti:

try:
    … (codice da eseguire se l'eccezione non si verifica)

except (<tipo_eccezione_1>, .. <tipo_eccezione_n>):
    … (codice da eseguire se un'eccezione di uno dei tipi specificati si verifica)

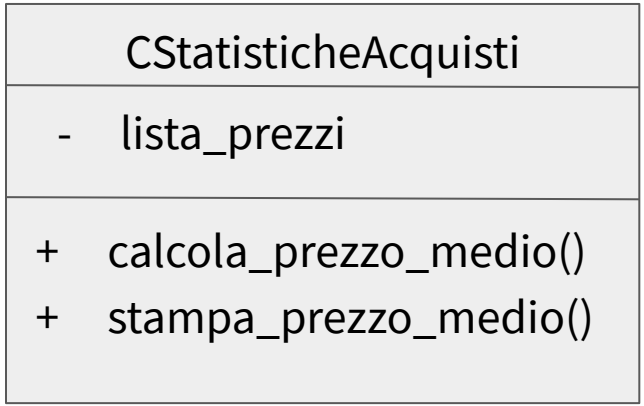
except (<tipo_eccezione_n+1>, .. <tipo_eccezione_z>):
    … (codice da eseguire se un'eccezione di uno dei tipi specificati si verifica)

Gestire Eccezioni

Ad esempio, supponiamo il gestore di un negozio abbia la necessità di avere una classe *StatisticheAcquisti* che gli permetta di ricevere, in fase di inizializzazione una lista (*lista_prezzi*) contenente i prezzi dei prodotti acquistati da un suo cliente e calcolare delle statistiche.

In particolare, questa classe deve avere due metodi pubblici:
-calcola_prezzo_medio che permette di calcolare il prezzo medio della lista memorizzata.
-stampa_prezzo_medio che permette di stampare a schermo il prezzo medio della lista memorizzata.

Gestire Eccezioni



Supponiamo anche di essere certi che quando viene invocato il metodo stampa_prezzo_medio, qualcuno sia davanti al terminale.

Gestire Eccezioni

```
class CStatisticheAcquisti():

  def __init__(self, lista_prezzi):
    self._lista_prezzi = lista_prezzi

  def calcola_prezzo_medio(self):
    prezzo_totale = 0
    for prezzo in self._lista_prezzi:
      prezzo_totale = prezzo_totale + prezzo
    return prezzo_totale / len(self._lista_prezzi)
  ...
```

Gestire Eccezioni

```
def stampa_prezzo_medio(self):
  prezzo_medio = self.calcola_prezzo_medio()
  print("Il prezzo medio è ", prezzo_medio)
```

Esempio:

```
oggetto_stat_acquisti = CStatisticheAcquisti([4, 8])
oggetto_stat_acquisti.stampa_prezzo_medio()
```

Stampa:

```
Il prezzo medio è  6.0
```

Gestire Eccezioni

Questo codice, se riceve una lista invalida genera un'eccezione.

```
oggetto_stat_acquisti = CStatisticheAcquisti(['a'])
oggetto_stat_acquisti.stampa_prezzo_medio()
...
TypeError: unsupported operand type(s) for +: 'int' and 'str'

oggetto_stat_acquisti = CStatisticheAcquisti([])
oggetto_stat_acquisti.stampa_prezzo_medio()
...
ZeroDivisionError: division by zero
```

Gestire Eccezioni

Supponiamo si voglia far si che, se viene inserita per sbaglio una lista invalida, ad esempio una lista vuota o una lista che contiene dei caratteri, il programma che richiama la funzione stampa_prezzo_medio non si blocchi ma stampi semplicemente a schermo il messaggio:
Non posso calcolare il prezzo medio per questa lista.

Gestire Eccezioni

Modifichiamo il metodo *stampa_prezzo_medio:*

```
def stampa_prezzo_medio(self):

  try:
     prezzo_medio = self.calcola_prezzo_medio()
     print("Il prezzo medio è ", prezzo_medio)

  except Exception:
     print("Non posso calcolare il prezzo medio per questa lista.")
```

Gestire Eccezioni

Dopo questa modifica, il seguente codice:

```
oggetto_stat_acquisti = CStatisticheAcquisti([4, 8])
oggetto_stat_acquisti.stampa_prezzo_medio()

oggetto_stat_acquisti = CStatisticheAcquisti([])
oggetto_stat_acquisti.stampa_prezzo_medio()

oggetto_stat_acquisti = CStatisticheAcquisti(['a'])
oggetto_stat_acquisti.stampa_prezzo_medio()
```

Gestire Eccezioni

Stamperà:

Il prezzo medio è 6.0
Non posso calcolare il prezzo medio per questa lista.
Non posso calcolare il prezzo medio per questa lista.

Gestire Eccezioni

Perchè modificare il metodo *stampa_prezzo_medio* e non modificare invece *calcola_prezzo_medio* così che qualsiasi codice che richiede il calcolo del prezzo medio e quindi richiama il metodo *calcola_prezzo_medio* non si blocchi?

NB: è una cattiva idea ma è importante capire per quale motivo lo è!

Gestire Eccezioni

```
class CStatisticheAcquisti():
  def __init__(self, lista_prezzi):
    self._lista_prezzi = lista_prezzi

  def calcola_prezzo_medio(self):
    try:
      prezzo_totale = 0
      for prezzo in self._lista_prezzi:
        prezzo_totale = prezzo_totale + prezzo
      return prezzo_totale / len(self._lista_prezzi)

    except Exception:
      print("Non posso calcolare il prezzo medio per questa lista.")
```

NB: è una cattiva idea ma è importante capire per quale motivo lo è!

Gestire Eccezioni

```
class CStatisticheAcquisti():
  def __init__(self, lista_prezzi):
    self._lista_prezzi = lista_prezzi

  def calcola_prezzo_medio(self):
    try:
      prezzo_totale = 0
      for prezzo in self._lista_prezzi:
        prezzo_totale = prezzo_totale + prezzo
      return prezzo_totale / len(self._lista_prezzi)

    except Exception:
      print("Non posso calcolare il prezzo medio per questa lista.")
```

NB: è una cattiva idea ma è importante capire per quale motivo lo è!

NB: nel caso in cui l'eccezione si verifichi, il metodo calcola_prezzo_medio non richiama la return e quindi resituisce None

Gestire Eccezioni

```
def stampa_prezzo_medio(self):
    prezzo_medio = self.calcola_prezzo_medio()
    if prezzo_medio is not None:
        print("Il prezzo medio è ", prezzo_medio)
```

Anche il metodo stampa_prezzo_medio va quindi modificato per tenere in considerazione il fatto che l'altro metodo potrebbe generare un'eccezione.

NB: è una cattiva idea ma è importante capire per quale motivo lo è!

Gestire Eccezioni

Questa soluzione potrebbe essere praticabile se il metodo *calcola_prezzo_medio* fosse privato e quindi fossimo sicuri che tutti i metodi che utilizzano quel metodo (nel nostro caso *stampa_prezzo_medio*) sono in grado di comportarsi in modo corretto nel caso in cui l'eccezione avvenga e venga gestita dal metodo *calcola_prezzo_medio*.

Poichè *calcola_prezzo_medio* è un metodo pubblico può potenzialmente venire invocato da qualsiasi codice quindi non abbiamo questa certezza!

E' quindi molto meglio far si che *calcola_prezzo_medio* generi l'eccezione e siano i metodi che lo utilizzano a gestirla.

Esercizio: Gestire Eccezioni

Considerate la classe *CScontrino* implementata come mostrata nella slide seguente.

Questa classe riceve al momento dell'inizializzazione la lista di prezzi dei prodotti acquistati dall'utente.

Inoltre ha un metodo pubblico che calcola il prezzo totale.

Esercizio: Gestire Eccezioni

```
class CScontrino:

  def __init__(self, lista_prezzi):
    self._lista_prezzi = lista_prezzi

  def calcola_totale(self):
    totale = 0
    for prezzo in self._lista_prezzi:
      totale = totale + prezzo
    return totale
```

Esercizio: Gestire Eccezioni

Per calcolare il totale questo metodo dovrà andare a sommare gli elementi presenti della lista.
Se ad esempio fosse stato erroneamente inserito un carattere invece che un numero verrebbe sollevata un'eccezione di tipo:

```
oggetto_scontrino = CScontrino([5,'a'])
totale = oggetto_scontrino.calcola_totale()
print(totale)
```

```
...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Esercizio: Gestire Eccezioni

Modificate quella funzione in modo da gestire, nel caso in cui si verifichi, l'eccezione mostrata, stampando a schermo il messaggio:
"Il totale di questo scontrino non può essere calcolato perchè la lista di prezzi inserita contiene uno o più prezzi invalidi".

Ed evitando così che il vostro programma di gestione del negozio si interrompa.

Nb: in questo caso stiamo supponendo che questo programma sia un programma che utilizzano i cassieri per creare gli scontrini quando i clienti effettuano gli acquisti, quindi stiamo supponendo di essere certi che ogni volta che quel metodo viene invocato ci sia un operatore davanti al terminale pronto per leggere il messaggio di errore.

Esercizio: Gestire Eccezioni

```
class CScontrino:
  def __init__(self, lista_prezzi):
    self._lista_prezzi = lista_prezzi

  def calcola_totale(self):
    totale = 0
    n_prezzi = len(self._lista_prezzi)
    try:
      for i in range(n_prezzi):
        totale = totale + self._lista_prezzi[i]
      return totale
    except TypeError:
      print("Il totale di questo scontrino non può essere calcolato " \
      "perchè la lista contiene uno o più prezzi invalidi.")
```

Esercizio: Gestire Eccezioni

```
oggetto_scontrino = CScontrino([5,'a'])
totale = oggetto_scontrino.calcola_totale()
if totale is not None:
  print(totale)
```

Verrà stampato:
Il totale di questo scontrino non può essere calcolato perchè la lista contiene uno o più prezzi invalidi.

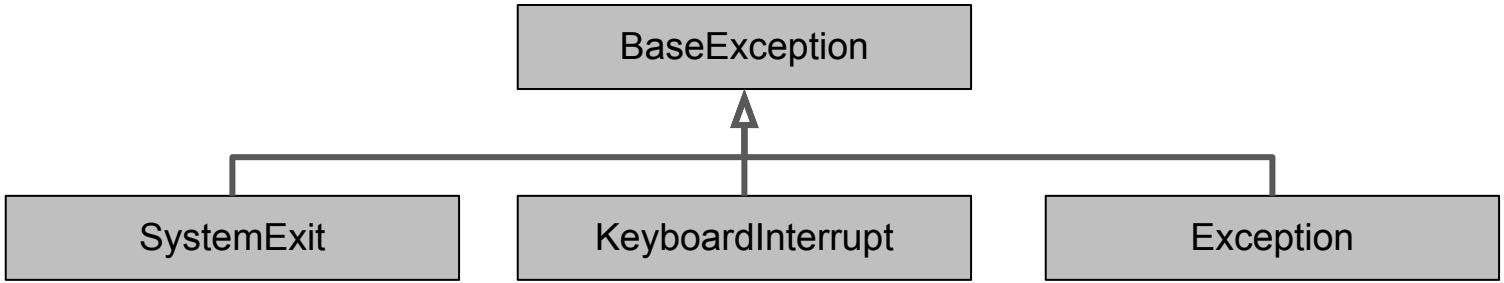
Definire Nuove Eccezioni

Le eccezioni sono oggetti.

Quando utilizziamo il codice per sollevare un'eccezione, utilizzando la sintassi:
raise <tipo_eccezione> ("messaggio per l'utente")
stiamo in realtà creando un oggetto appartenente alla classe <tipo_eccezione>
e passando al metodo inizializzatore il messaggio per l'utente.

E' quindi possibile definire nuove eccezioni ereditando dalle classi esistenti per la gestione delle eccezioni.

Definire Nuove Eccezioni



- **SystemExit** sono le eccezioni che vengono sollevate quando il programma termina naturalmente (senza errori).
- **KeyboardInterrupt** sono le eccezioni che vengono sollevate quando il programma viene terminato dall'utente con una combinazione di tasti (es: CTRL+C).
- **Exception** sono tutte le altre eccezioni.

Definire Nuove Eccezioni

Per definire una nuova eccezione si crea una classe che eredita dalla classe Exception.

Ad esempio se volessimo creare un'eccezione per un programma di gestione di un bancomat chiamata PrelievoInvalido potremmo semplicemente definirla come:

```
class CPrelievoInvalido(Exception):
  pass
```

Definire Nuove Eccezioni

Poichè eredita dalla classe Exception, quando viene sollevata, ad esempio dal codice:

```
raise CPrelievoInvalido("Il prelievo richiesto non può essere effettuato")
```

Come per le altre eccezioni, il programma termina e viene stampato a schermo il messaggio di errore.

Es:

```
...
__main__.CPrelievoInvalido: Il prelievo richiesto non può essere effettuato
```

Definire Nuove Eccezioni

Potenzialmente si può creare anche classi di eccezione più complesse, ad esempio per fornire un messaggio più informativo.

Ad esempio, supponiamo di voler modificare la classe creata precedentemente in modo da far sì che stampi sempre a schermo il messaggio:
Il prelievo richiesto non può essere effettuato. Hai cercato di prelevare x euro più di quelli presenti sul tuo conto.

Nb: il testo del messaggio vogliamo sia sempre lo stesso, mentre deve cambiare il valore di x mostrato nel messaggio.

Definire Nuove Eccezioni

```
class CPrelievoInvalido(Exception):

  def __init__(self, importo_sul_conto, importo_prelevato):

     messaggio = "Il prelievo richiesto non può essere effettuato. "
     Importo_non_presente  = importo_prelevato - importo_sul_conto
     messaggio = messaggio + "Hai cercato di prelevare " + str( importo_non_presente) + \
                          " euro più di quelli presenti sul tuo conto."

     super().__init__(messaggio)
```

Definire Nuove Eccezioni

```
raise CPrelievoInvalido(100, 150)
```

Stampa:

```
__main__.CPrelievoInvalido: Il prelievo richiesto non può essere effettuato. Hai cercato di prelevare 50 euro più di quelli presenti sul tuo conto.
```

Esercizio: definire Nuove Eccezioni

Definire una nuova eccezione per un programma di gestione di un magazzino. La nuova classe dovrà chiamarsi CProdottiNonSufficienti e dovrà stampare a schermo il messaggio:

“Il numero di prodotti richiesto non è presenti in magazzino. In magazzino sono presenti solo x prodotti del tipo richiesto”.

Dove x è il numero di prodotti presenti in magazzino che deve essere passato come argomento al metodo inizializzatore dell’oggetto.

Esercizio: definire Nuove Eccezioni

```
class CProdottiNonSufficienti(Exception):

  def __init__(self, n_prodotti_in_magazzino, n_prodotti_richiesti):

    messaggio = "Il numero di prodotti richiesto non è presenti in magazzino. " \
          "In magazzino sono presenti solo "
    prodotti_non_presenti = n_prodotti_richiesti - n_prodotti_in_magazzino
    messaggio = messaggio + str(prodotti_non_presenti) + \
          " prodotti del tipo richiesto."

    super().__init__(messaggio)
```

Esercizio: definire Nuove Eccezioni

```
raise CProdottiNonSufficienti(100, 150)
```

```
__main__.CProdottiNonSufficienti: Il numero di prodotti richiesto non è presenti in magazzino. In magazzino sono presenti solo 50 prodotti del tipo richiesto.
```