Pattern Recognition
and Applications Lab

La Programmazione ad Oggetti in Python

Docente: Ambra Demontis

Anno Accademico: 2021 - 2022

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:
- alcune caratteristiche degli oggetti di tipo stringa
- le espressioni regolari

Selezione delle stringhe.

Le stringhe possono essere viste come delle sequenze di caratteri.

E’ possibile selezionare una parte di una stringa utilizzando gli operatori di selezione.

```
stringa = "LPO - Python3"
prima_parte_stringa = stringa[:3]
print(prima_parte_stringa)
```

Stampa:
LPO

Selezione delle stringhe.

La sequenza di caratteri \n è riconosciuta da Python come il carattere “a capo”.

Esempio:

```
stringa = "LPO\nPython3"
print(stringa)
```

Stampa:

LPO
Python3

Il Metodo Format

Le stringhe sono oggetti. In quanto tali, implementano diversi metodi utili.

Uno di questi metodi è il metodo format.

Fino ad ora per costruire una stringa composta da i valori di alcune variabili abbiamo utilizzato l'operatore di concatenazione...

Tuttavia, dovendo inserire in una stringa diversi valori il codice risulta abbastanza lungo e poco leggibile.

Il Metodo Format

Supponete di avere tre variabili (nome, cognome, voto) e di voler stampare a schermo la stringa “Il voto conseguito da <nome> <cognome> è <voto>”.

Utilizzando l’operatore di concatenazione, il codice sarebbe:

```
stringa = "Il voto conseguito da " + nome + " " + cognome + " è " + str(voto)
```

Il Metodo Format

Il metodo format ci permette di utilizzare una sintassi più compatta.

Grazie a questo metodo possiamo creare direttamente un’unica stringa inserendo i caratteri {} nei punti nei quali vogliamo inserire i valori.
Es: "Il voto conseguito da {} {} è {}"

Richiamando il metodo format possiamo poi sostituire ai caratteri {} i valori che vogliamo.

Il Metodo Format

```
nome = "Anna"
cognome = "Bianchi"
voto = 28

stringa = "Il voto conseguito da {} {} è {}".format(nome, cognome, voto)

print(stringa)
```

Utilizzando questo metodo non è necessario convertire i valori numerici (es, voto) in stringa. Vengono automaticamente convertiti dal metodo stesso.

Il Metodo Format

Grazie a questo metodo, possiamo anche facilmente scegliere il formato con il quale i valori verranno inseriti nella stringa.

Supponete di avere un numero con tante cifre decimali e di volerne mostrare solo alcune.

In quel caso, possiamo usare la sintassi:
{ : . <numero_decimali_da_mostrare> f }

Il Metodo Format

Esempio:

```
altezza = 35.555556
stringa_altezza = "{:.2f}".format(altezza)
print(stringa_altezza)
```

Stamperà:

```
35.56
```

Il Metodo Format

Per i numeri interi* possiamo usare la sintassi:
{ : d}

Oppure, se vogliamo vengano rappresentati come numeri con la virgola, la stessa sintassi che utilizziamo per i numeri con la virgola:

{ : . <numero_decimali_da_mostrare> f }

*questa sintassi può essere utilizzata solo per i numeri interi, se tentassimo di utilizzarla con quelli con la virgola verrebbe sollevata un'eccezione.

Il Metodo Format

Esempio:

```
altezza = 35

stringa_altezza = "{:.2f}".format(altezza)
print(stringa_altezza)

stringa_altezza = "{:d}".format(altezza)
print(stringa_altezza)
```

Stamperà:

```
35.00
35
```

Il Metodo Split

Un’altro metodo spesso utilizzato dai programmatori è il metodo split.

Il metodo split delle stringhe serve a dividere la stringa, in base ad un carattere scelto, in una lista di sottoparti.

Ad esempio, a dividere stringa: "LPO Python3" in base al carattere spazio ottenendo la lista di sottoparti: ["LPO", e "Python3"].

Il Metodo Split

La sintassi del metodo split è:
<oggetto_stringa>.split(“carattere_scelto”)

Quindi nel caso di esempio, potremmo scrivere:

```
stringa = "LPO Python3"
stringa_divisa = stringa.split(" ")
print(stringa_divisa)
```

Stampa:
['LPO', 'Python3']

Esercizi sul Metodo Format e Split

Sul metodo format:
Supponete di avere una variabile chiamata “lunghezza” che contiene il valore 44.1233.
Utilizzare il metodo format per stampare a schermo, il valore contenuto nella variabile lunghezza mostrando solo le cifre intere e le prime 3 cifre decimali.

Sul metodo split:
Utilizzare il metodo split per suddividere la stringa: "LPO_Python_3" utilizzando come separatore il carattere underscore.

Esercizi sul Metodo Format e Split

Soluzione esercizio sul metodo format:

```
lunghezza = 44.1233
stringa_lunghezza = "{:.3f}".format(lunghezza)
print(stringa_lunghezza)
```

Stamperà:
44.123

Esercizi sul Metodo Format e Split

Soluzione esercizio sul metodo split:

```
stringa = "LPO_Python_3"
stringa_divisa = stringa.split("_")
print(stringa_divisa)
```

Stamperà:
['LPO', 'Python', '3']

Le Espressioni Regolari

Un programma può dover **analizzare delle stringhe**.

Questo **può essere necessario per**:

- controllare che rispettino un determinato formato
- estrarre informazioni utili dalle stesse

Ad esempio può essere necessario:

- controllare se una stringa contiene un URL valida
- estrarre data e orario di tutti i messaggi di warning presenti in un log file

Le Espressioni Regolari

A quali figure professionali capita spesso di dover analizzare stringhe?

- Web developer
- Amministratori di sistema
- Amministratori di rete
- A chi deve effettuare Natural Language Processing (e.g., capire qual è l'argomento che viene trattato in un testo).

  ...

Le Espressioni Regolari

In molti linguaggi di programmazione l'**analisi delle stringhe** viene effettuata utilizzando **espressioni regolari**.

Queste costituiscono un **mini-linguaggio** con una sintassi creata ad-hoc per effettuare l'analisi delle stringhe.

Le Espressioni Regolari

Perché l'analisi delle stringhe viene effettuata con questo mini-linguaggio ad-hoc invece che con delle classi create per questo scopo?

Ci sono stati dei tentativi, pubblicati in articoli scientifici, di modellare questo problema seguendo il paradigma della programmazione ad oggetti.

Tuttavia, i metodi proposti producono del codice molto verboso e difficile da leggere, pertanto non sono molto utilizzati.

Le Espressioni Regolari

Anche in Python l'analisi delle stringhe viene effettuata utilizzando espressioni regolari.

Tuttavia, fornisce alcune classi e oggetti che possono essere utilizzati per costruire ed eseguire espressioni regolari.

In Python, le funzionalità per l'utilizzo delle espressioni regolari sono messe a disposizione dalla libreria "re".

Le Espressioni Regolari

Supponiamo di voler controllare se una stringa è identica ad un'altra.

Ad esempio, di voler controllare se una stringa data è identica alla stringa: "session opened".

Per far questo possiamo utilizzare **la funzione match**, definita dalla libereria re.

Le Espressioni Regolari

La funzione match **controlla se una “stringa da cercare” può essere sovrapposta ad una “stringa data”**.

Nel caso in cui questo sia possibile, la funzione match restituisce un oggetto, altrimenti restituisce None.

Le Espressioni Regolari

Nel caso in cui siano identiche questo è ovviamente vero.

```
import re
stringa_data = "session opened"
espressione_regolare = "session opened" # stringa da cercare
presente = re.match(espressione_regolare, stringa_data)
print(presente)
```

Stampa:

<re.Match object; span=(0, 14), match='session opened'>

Le Espressioni Regolari

Cosa succede nel caso in cui non siano identiche?

Ricordiamo che la funzione match verifica se la stringa da cercare può essere sovrapposta all’inizio della stringa data.

Le Espressioni Regolari

espressione_regolare = "session opened" *# stringa da cercare*

stringa_data = "session opened" match? Si!

stringa_data = "session" match? No

stringa_data = "opened" match? No

stringa_data = "xx session opened" match? No

stringa_data = "session opened for user root" match? Si!

Le Espressioni Regolari

L’oggetto che viene restituito quando la sovrapposizione viene identificata ha un metodo chiamato span che se invocato restituisce una tupla con l’indice di inizio e di fine della sovrapposizione.

```
espressione_regolare = "session opened" # stringa da cercare
stringa_data = "session opened for user root"
print( re.match(espressione_regolare, stringa_data).span() )
```

Stampa:

(0, 14)

Le Espressioni Regolari

Come facciamo per far si che la funzione match restituisca un oggetto solo se la stringa cercata è identica alla stringa data?

Le Espressioni Regolari

Le espressioni regolari possono contenere dei simboli che vengono interpretati in modo particolare.

Ad esempio:

Il simbolo **^** rappresenta l'inizio della stringa

Il simbolo **$** rappresenta la fine della stringa

Le Espressioni Regolari

Come facciamo per far si che la funzione match restituisca un oggetto solo se la stringa cercata è identica alla stringa data?

Possiamo sfruttare il simbolo **$**

Scrivendo l'espressione regolare così:

espressione_regolare = "session opened$"

In questo modo stiamo dicendo alla funzione match che non vogliamo solo che la stringa data inizi con "session open" ma anche che dopo questi caratteri termini.

Le Espressioni Regolari

espressione_regolare = "session opened$" *# stringa da cercare*

espressione_regolare = "session opened" match? Si!

stringa_data = "session" match? No

stringa_data = "opened" match? No

stringa_data = "xx session opened" match? No

stringa_data = "session opened for user root" match? No!

Le Espressioni Regolari

Nell’esempio precedente abbiamo utilizzato la funzione match per cercare dei caratteri precisi (nel caso di esempio quelli che compongono la stringa “session opened”).

Tuttavia **può essere utilizzata anche per cercare dei caratteri scelti arbitrariamente da un insieme di caratteri possibili**...

Le Espressioni Regolari

Supponiamo di voler far si che la funzione match ci restituisca un oggetto se trova una stringa composta dalla stringa “dati_studenti_versione” seguita da **un carattere qualsiasi**.

Quindi la stringa che vogliamo cercare contiene un carattere arbitrario.

Per rappresentare un carattere “qualsiasi” possiamo utilizzare il simbolo **.**

Le Espressioni Regolari

espressione_regolare = "dati_studenti_versione.$" *# stringa da cercare*

stringa_data = "dati_studenti_versione3" match? Si!

stringa_data = "dati_studenti_versione" match? No! (manca il carattere qualsiasi)

Le Espressioni Regolari

Supponiamo di voler far si che la funzione match ci restituisca un oggetto se trova una stringa composta dalla stringa “dati_studenti_versione” seguita non più da un carattere qualsiasi ma da un **carattere a scelta tra**: 1, 2, e 3.

Possiamo scrivere i caratteri possibili racchiusi tra parentesi quadre.

Es: [123]

Le Espressioni Regolari

```
espressione_regolare = "dati_studenti_versione[123]$" # stringa da cercare

stringa_data = "dati_studenti_versione3"          match? Si!

stringa_data = "dati_studenti_versione"           match? No

stringa_data = "dati_studenti_versione4"          match? No
```

Le Espressioni Regolari

Se i caratteri possibili sono sequenziali possiamo scrivere solo il primo e l'ultimo separati dal carattere trattino -

Es: [1-3]

Questa notazione può esserci molto utile se vogliamo ad esempio:

[0-9] una qualsiasi cifra compresa tra zero e 9

[a-z] un qualsiasi carattere tra le lettere minuscole

[A-Z] un qualsiasi carattere tra le lettere maiuscole

Le Espressioni Regolari

Potrebbe capitarci di volere che il carattere a scelta possa appartenere a due o più insiemi differenti.

In quel caso possiamo semplicemente scriverli affiancati all'interno delle parentesi quadre.

Ad esempio, se volessimo che il carattere sia una lettera qualsiasi e che possa essere sia maiuscola che minuscola scriveremo:

[a-zA-Z]

Le Espressioni Regolari

espressione_regolare = "dati_studenti_versione[a-zA-Z]$" *# stringa da cercare*

stringa_data = "dati_studenti_versionea" match? Si!

stringa_data = "dati_studenti_versioneB" match? Si!

stringa_data = "dati_studenti_versione4" match? No!

Le Espressioni Regolari

Come abbiamo visto il carattere . nelle espressioni regolari rappresenta un carattere qualsiasi.

Come si fa quindi a cercare una stringa che contiene esattamente il carattere punto? Ad esempio la stringa: dati_studenti_versione3.txt ?

(Scrivendo l'espressione regolare come: dati_studenti_versione3.txt$ verrebbe restituito un oggetto per tutte quelle stringhe che hanno un carattere qualsiasi al posto del punto).

Le Espressioni Regolari

Si utilizza il carattere backslash \

espressione_regolare = "dati_studenti_versione\.txt$" *# stringa da cercare*

stringa_data = "dati_studenti_versione.txt" match? Si!

stringa_data = "dati_studenti_versione3txt" match? No

Esercizio sulle Espressioni Regolari

Acquisire in input una stringa e verificare se è esattamente uguale all'URL composta da:

La stringa "https://lpo/slide_parte_ " seguita da un numero qualsiasi e poi dalla stringa ".pdf"

Esercizio sulle Espressioni Regolari

```
import re

espressione_regolare = "https://lpo/slide_parte_[0-9]\.pdf$"
stringa_data = "https://lpo/slide_parte_3.pdf"

print( re.match(espressione_regolare, stringa_data) )
```

Stampa:

<re.Match object; span=(0, 29), match='https://lpo/slide_parte_3.pdf'>

Le Espressioni Regolari

Può capitare di non conoscere a priori il numero di caratteri dai quali la stringa deve essere costituita.

Supponete di voler controllare se la stringa data è composta dalla stringa "idx_" seguita da una stringa di lunghezza qualsiasi.

Le Espressioni Regolari

Per far si che un carattere debba essere ripetuto un numero di volte qualsiasi possiamo mettere alla sua destra il simbolo *****

Nb: il numero di volte può essere anche pari a zero.

Ad esempio se vogliamo indicare un carattere qualsiasi ripetuto un numero di volte qualsiasi dobbiamo scrivere

`.*`

Le Espressioni Regolari

espressione_regolare = "idx_.*" *# stringa da cercare*

stringa_data = "idx_ABC55"	match? Si!
stringa_data = "idx_JU(/(AAA"	match? Si!
stringa_data = "idx_"	match? Si!
stringa_data = "x_"	match? No

Le Espressioni Regolari

Supponete di voler controllare se la stringa data è composta dalla stringa "idx_" seguita da una stringa di lunghezza qualsiasi composta da lettere maiuscole e numeri.

Possiamo far precedere al simbolo ***** l'indicazione dei caratteri ammissibili.

Es:

[A-Z0-9]*

Le Espressioni Regolari

espressione_regolare = "idx_[A-Z0-9]*$" *# stringa da cercare*

stringa_data = "idx_ABC55" match? Si!

stringa_data = "idx_JU(/(AAA" match? No!

Nb: se non utilizzassimo il simbolo $ verrebbe trovato un match anche qui.

stringa_data = "idx_" match? Si!

stringa_data = "x_" match? No

Esercizio sulle Espressioni Regolari

Create un’espressione regolare che, utilizzando la funzione match, vi permetta di controllare se la stringa data è composta dalla stringa "idx_" seguita da una stringa composta da lettere maiuscole e numeri lunga almeno un carattere.

Esercizio sulle Espressioni Regolari

```
import re

espressione_regolare = "idx_[A-Z0-9]*[A-Z0-9]$"
stringa_data = "idx_4AC5"

print( re.match(espressione_regolare, stringa_data) )
```

Stampa:

<re.Match object; span=(0, 8), match='idx_4AC5'>

Le Espressioni Regolari

Se vogliamo che un **carattere** debba essere **ripetuto** esattamente **n volte** possiamo fargli succedere il numero di volte indicato tra **parentesi graffe**.

Es:

a{3} rappresenta aaa

.{3} rappresenta tre caratteri qualsiasi

Le Espressioni Regolari

Supponete di voler controllare se la stringa data è composta dalla stringa "idx_" seguita da una stringa di 3 caratteri che possono comprendere solo lettere maiuscole o numeri.

espressione_regolare = "idx_[A-Z0-9]{3}$" *# stringa da cercare*

stringa_data = "idx_ABC55" match? No

stringa_data = "idx_ABC" match? Si!

stringa_data = "idx_AB5" match? Si!

Le Espressioni Regolari

E’ anche possibile volere che un intero gruppo di caratteri possa essere ripetuto esattamente n volte.

In quel caso, possiamo **raggruppare** il gruppo di **caratteri** dentro delle **parentesi tonde.** In quel caso, verrà considerato come un carattere unico.

Es:

(ab){3} rappresenta ababab

Le Espressioni Regolari

Se volessimo capire se è presente una sequenza o un'altra sequenza di caratteri possiamo utilizzare sempre le parentesi tonde e il carattere | per indicare che può essere presente uno qualsiasi dei gruppi separati dal carattere |.

Es: (12 | 54)

Il match verrà trovato se è presente 12 oppure 54

Le Espressioni Regolari

espressione_regolare = "idx_(12|54|aa)$" *# stringa da cercare*

stringa_data = "idx_12"	match? Si!
stringa_data = "idx_54"	match? Si!
stringa_data = "idx_aa"	match? Si!
stringa_data = "idx_ab"	match? No

Le Espressioni Regolari

Fino ad ora abbiamo visto come utilizzare le espressioni regolari per controllare se una stringa data rispetta o no un particolare formato.

Possono però essere utili anche per **estrarre delle informazioni da una stringa.**

Le Espressioni Regolari

Supponete di lavorare in un'azienda nel quale tutti gli indirizzi email hanno il formato:

<nome>.<cognome>@xpy.com

Dove nome e cognome sono composti da lettere minuscole.

Supponete di voler, quando vi arriva una mail da un vostro collega, estrarre dall'indirizzo email del mittente il suo nome e il suo cognome.

Le Espressioni Regolari

Se volessimo semplicemente controllare se la mail del nostro collega ha il formato previsto l'espressione regolare sarebbe:

```
espressione_regolare = "[a-z]*\.[a-z]*@xpy.com$" # stringa da cercare

stringa_data = "anna.bianchi@xpy.com"          match? Si!

stringa_data = "anna.bianchi@gmail.com"        match? No
```

Le Espressioni Regolari

Per estrarre delle sottoparti possiamo:

1) Indicare tra parentesi tonde le sottoparti che vogliamo estrarre
2) **Utilizzare il metodo *groups* dell'oggetto** che viene restituito nel caso in cui ci sia il matching (la stringa rispetti il formato).

Il metodo groups ci restituirà una tupla contenente tutte le sottoparti della stringa corrispondenti alle parti dell'espressione regolare indicate tra parentesi.

Le Espressioni Regolari

L’espressione regolare che avevamo era:

espressione_regolare = "[a-z]*\.[a-z]*@xpy.com$"

Dobbiamo inserire le parentesi racchiudendo le parti dell’espressione regolare che rappresentano le parti della sottostringa che vogliamo andare ad estrarre:

espressione_regolare = "([a-z]*)\.([a-z]*)@xpy.com$"

Le Espressioni Regolari

Poi utilizziamo la funzione groups dell'oggetto che ci viene restituito nel caso in cui ci sia matching.

```
espressione_regolare = "([a-z]*)\.([a-z]*)@xpy.com$" # stringa da cercare
stringa_data = "anna.bianchi@xpy.com"
match_obj = re.match(espressione_regolare, stringa_data)
if match_obj is not None:
  nome = match_obj.groups()[0]
  cognome = match_obj.groups()[1]
```

Esercizio sulle Espressioni Regolari

Supponete di lavorare per un'azienda che offre diversi servizi, tra i quali una casella email gratuita ma con uno spazio di archiviazione ridotto e ampliabile sottoscrivendo un canone mensile.

La vostra azienda offre anche altri servizi e per accedervi è necessario registrarsi sul vostro sito web fornendo un indirizzo email.

Decidete quindi di estrarre dagli indirizzi email con i quali le persone si registrano sul sito web, il nome della compagnia che fornisce l'indirizzo email in modo da effettuare delle statistiche per capire chi sono i vostri maggiori rivali.

Esercizio sulle Espressioni Regolari

Ad esempio, se l’indirizzo email del vostro cliente fosse:

"anna.bianchi@tiscali.it"

Voi vorreste estrarre la stringa "tiscali".

Esercizio sulle Espressioni Regolari

```
import re
espressione_regolare = "[a-z\._]*@([a-z]*)\.[a-z]*$"
stringa_data = "anna.bianchi@tiscali.it"
match_obj = re.match(espressione_regolare, stringa_data)

if match_obj is not None:
  azienda_rivale = match_obj.groups()[0]
print("azienda rivale: ", azienda_rivale)
```

(Supponendo l'indirizzo email possa essere composto da lettere, punti o underscore.)

Stampa:

azienda rivale: tiscali