

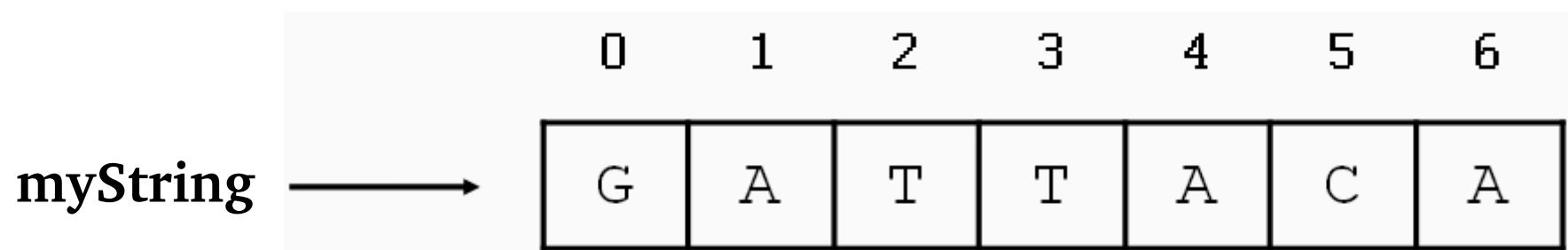
# **Text Processing in Python**

# Text Processing in Python

- Python è un linguaggio molto potente per la manipolazione e la gestione di informazioni testuali
- In questa lezione:
  - Ripassi e approfondimenti sulle stringhe: semplici operazioni sul testo
  - Espressioni regolari: un potente strumento di text processing, anche in Python

# Rappresentazione di stringhe

- Ogni stringa è memorizzata nella memoria del calcolatore come una lista di caratteri
- `>>> myString = "GATTACA"`



# Accesso ai singoli caratteri

- E' possibile accedere ai singoli caratteri utilizzando gli indici tra parentesi quadre

```
>>> myString = "GATTACA"
```

```
>>> myString[0]
```

'G'

```
>>> myString[1]
```

'A'

```
>>> myString[-1]
```

'A'

```
>>> myString[-2]
```

'C'

```
>>> myString[7]
```

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

  IndexError: string index out of range

Gli indici negativi iniziano dalla fine della stringa e crescono verso sinistra

# Accesso alle sottostringhe

```
>>> myString = "GATTACA"
```

```
>>> myString[1:3]
```

```
'AT'
```

```
>>> myString[:3]
```

```
'GAT'
```

```
>>> myString[4:]
```

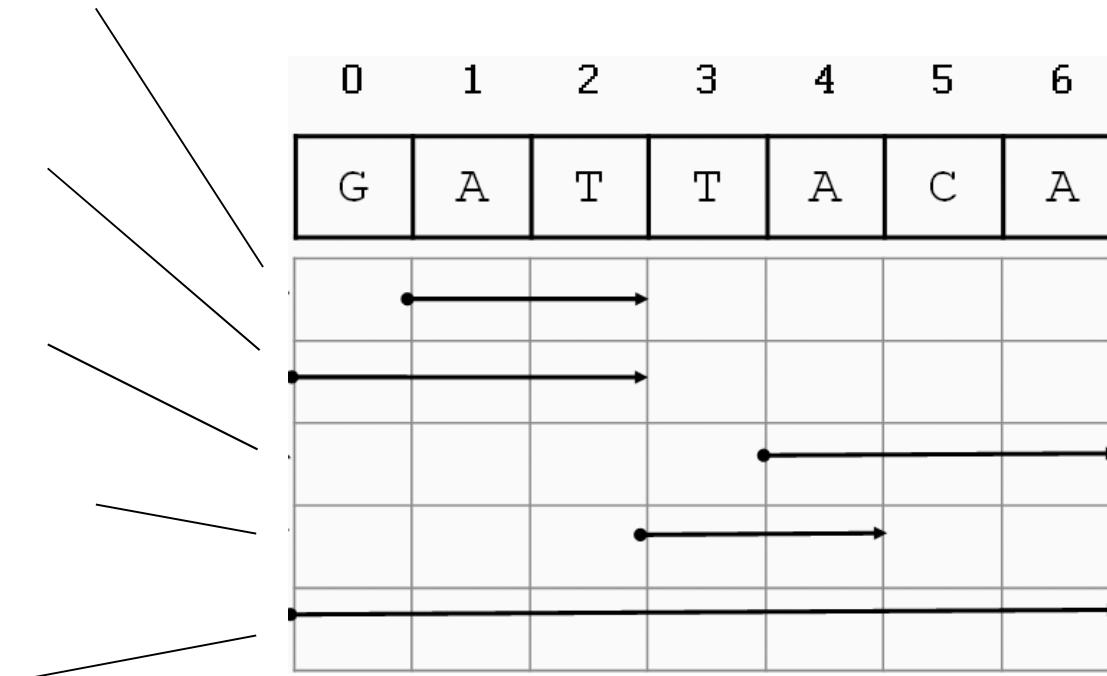
```
'ACA'
```

```
>>> myString[3:5]
```

```
'TA'
```

```
>>> myString[:]
```

```
'GATTACA'
```



# Caratteri speciali

- Per introdurre un carattere speciale si utilizza il **backslash**

```
>>> "He said, "Wow!""  
      File "<stdin>", line 1  
          "He said, "Wow!""  
              ^  
SyntaxError: invalid syntax  
>>> "He said, 'Wow!'"  
"He said, 'Wow!'"  
>>> "He said, \"Wow!\\""  
'He said, "Wow!"'
```

Sequenza  
di escape

\\"

\

\"

\n

\t

Significato

**Backslash**

**Single  
quote**

**Double  
quote**

**Newline**

**Tab**

# Alcuni operatori utili

```
>>> len("GATTACA")
7
>>> "GAT" + "TACA"
'GATTACA'
>>> "A" * 10
'AAAAAAAAAA
>>> "GAT" in "GATTACA"
True
>>> "AGT" in "GATTACA"
False
```

- Lunghezza
- Concatenazione
- Ripetizione
- Test di sottostringa

# Alcuni esempi di metodi stringa

• ESEMPI:  
string.py

```
>>> "GATTACA".find("ATT")
1
>>> "GATTACA".count("T")
2
>>> "GATTACA".lower()
'Gattaca'
>>> "Gattaca".upper()
'GATTACA'
>>> "GATTACA".replace("G", "U")
'UATTACA'
>>> "GATTACA".replace("C", "U")
'GATTAUA'
>>> "GATTACA".replace("AT", "***")
'G***TACA'
>>> "GATTACA".startswith("G")
True
>>> "GATTACA".startswith("g")
False
```

# Split e join

▪ ESEMPI:  
splitjoin.py

- Il metodo **split()** è utilizzato per suddividere una stringa in una sequenza di elementi

```
>>> '1+2+3+4+5'.split('+')
```

```
['1', '2', '3', '4', '5']
```

```
>>> 'Using the default'.split()
```

```
['Using', 'the', 'default']
```

- Il metodo **join()** è utilizzato per unire una sequenza di stringhe

```
>>> seq = ['1', '2', '3', '4', '5']
```

```
>>> sep = '+'
```

```
>>> sep.join(seq)
```

```
'1+2+3+4+5'
```

# strip

- Il metodo **strip (s[, chars])** restituisce una copia della stringa con i caratteri iniziali e finali rimossi
  - Se **chars** non è specificato, vengono rimossi gli spazi bianchi (tab, spazio)
  - Utile per “ripulire” le stringhe

```
>>> ' spacious '.strip()  
'spacious'  
>>> 'www.example.com'.strip('cmowz.')  
'example'
```

# Le stringhe sono immutabili

- Attenzione: le stringhe non possono essere modificate
- Per la modifica è necessario crearne di nuove

```
>>> s = "GATTACA"
```

```
>>> s[3] = "C"
```

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

    TypeError: object doesn't support item assignment

```
>>> s = s[:3] + "C" + s[4:]
```

```
>>> s
```

'GATCACA'

```
>>> s = s.replace("G","U")
```

```
>>> s
```

'UATCACA'

# Le stringhe sono immutabili

- **Attenzione: i metodi stringa non modificano la stringa, ma ne restituiscono una nuova**

```
>>> sequence = "ACGT"  
>>> sequence.replace("A", "G")  
'GCGT'  
>>> print sequence  
ACGT
```

```
>>> sequence = "ACGT"  
>>> new_sequence = sequence.replace("A", "G")  
>>> print new_sequence  
GCGT
```

# Espressioni regolari

- **Le espressioni regolari sono un potente mezzo di elaborazione del testo**
- **Python include un ricco pacchetto per la gestione di espressioni regolari: re**
  - **Gestione delle espressioni in stile Perl**
  - **Creazione di un oggetto rappresentante l'espressione regolare**
  - **Invocazione di metodi per il match di espressioni regolari su stringa**

# Sintassi di base

- Una stringa di testo regolare fa match con se stessa
  - “test” fa match in “Questo è un test”
- “.” fa match con ogni carattere singolo
- $x^*$  fa match con zero o più x
  - “ $a^*$ ” fa match con ”, ’a’, ’aa’, etc.
- $x^+$  fa match con una o più x
  - “ $a^+$ ” fa match con ’a’, ’aa’, ’aaa’, etc.
- $x^?$  fa match con zero o una x
  - “ $a^?$ ” fa match con ” o ’a’
- $x\{m, n\}$  fa match con i x, dove  $m < i < n$ 
  - “ $a\{2,3\}$ ” fa match con ’aa’ o ’aaa’

# Sintassi di base

- **[x]** fa match con un qualunque carattere della lista x
  - “[abc]” fa match con ‘a’, ‘b’ o ‘c’
- **[^x]** fa match con un qualunque carattere non incluso in x
  - “[^abc]” fa match con qualunque carattere eccetto ‘a’, ‘b’, o ‘c’
- Le parentesi () sono utilizzate per i gruppi
  - “(abc)+” fa match con ‘abc’, ‘abcabc’, ‘abcababcabc’, etc.
- **x|y** fa match con x o y
  - “this|that” fa match con ‘this’ e ‘that’, ma non ‘thisthat’.

# Sintassi di base

- “\d” fa match con una cifra
- “\D” fa match con una non-cifra
- “\s” fa match con un carattere whitespace
- “\S” fa match con un carattere non-whitespace
- “\w” fa match con un carattere alfanumerico
- “\W” fa match con un carattere non alfanumerico
- “^” fa match con l'inizio della stringa
- “\$” fa match con la fine della stringa
- “\b” fa match con un word boundary
- “\B” fa match con una posizione che non è un word boundary

# Escaping di caratteri speciali

- **Attenzione ai caratteri speciali!**
  - 'python.org' fa match con 'python.org' ma anche con 'pythonzorg', etc.
- **Per fare in modo che un carattere speciale si comporti da carattere normale, occorre farne l'escaping (carattere di backslash anteposto):**
  - 'python\\l.org' fa match con 'python.org'
- **Attenzione al doppio backslash (non singolo)!**
  - **Nelle stringhe normali, ogni backslash viene valutato**
- **Per far arrivare al modulo re la stringa corretta (python\\l.org) occorre annullare il primo backslash**

# Escaping di caratteri speciali

- Per quanto riguarda gli insiemi di caratteri, l'escaping è possibile ma non necessario
- Attenzione però ai seguenti casi:
  - E' necessario l'escaping di ^ se compare all'inizio del set e non si vuole intenderlo come operatore di negazione.
  - Analogamente, ] e - devono essere posti all'inizio del set oppure deve esserne fatto l'escaping

# Stringhe “raw”

- Una pratica alternativa all’escaping all’interno di stringhe è di utilizzare le raw string
- Una stringa raw non è soggetta ad escaping
- Una stringa raw ha una r anteposta
  - ‘Questa è una stringa’
  - r‘Questa è una stringa raw’
- Differenze con le stringhe normali

```
>>> print 'C:\\nowhere'
```

C:\\nowhere

- Con una stringa raw:

```
>>> print r'C:\\nowhere'
```

C:\\nowhere

# Esempi di utilizzo di stringhe “raw”

- Supponiamo di voler scrivere una RE che indichi le corrispondenze della stringa "\section", tipico comando LATEX
- Senza utilizzare stringhe raw, dovremmo seguire i seguenti passi:
  - \section Stringa di testo di cui cercare la corrispondenza
  - \\section Annullamento del backslash per il modulo re
  - "\\\\\\section" Annullamento del backslash nella stringa normale

# Esempi di utilizzo di stringhe “raw”

- utilizzando stringhe raw, l'esempio precedente si riduce a:
  - `\section` Stringa di testo di cui cercare la corrispondenza
  - `\\"section` Annullamento del backslash per il modulo re
  - `r'\\section'` Uso della stringa raw

# modulo re: alcune funzioni importanti

- **search(pattern, string[, flags])**
  - Effettua la ricerca di pattern in string
- **match(pattern, string[, flags])**
  - Fa match con pattern all'inizio di string
- **split(pattern, string[, maxsplit=0])**
  - Suddivide string in base alle occorrenze di pattern
- **findall(pattern, string)**
  - Restituisce una lista di tutte le occorrenze di pattern in string

# modulo re: alcune funzioni importanti

- **sub(pat, repl, string[, count=0])**
  - **Sostituisce con repl tutte le occorrenze di pat in string**
- **compile(pattern[, flags])**
  - **Crea un oggetto pattern da una stringa con una regexp**
- **escape(string)**
  - **Effettua l'escaping di tutti i caratteri speciali di string**

# Search e Match

- I due metodi principali sono `re.search()` e `re.match()`
  - `re.search()` ricerca un pattern ovunque nella stringa
  - `re.match()` effettua la ricerca solo a partire dall'inizio della stringa
- Tali metodi restituiscono `None` se il pattern non è trovato e un “match object” se lo è

```
>>> pat = "a*b"  
>>> import re  
>>> re.search(pat,"fooaaabcde")  
<_sre.SRE_Match object at 0x809c0>  
>>> re.match(pat,"fooaaabcde")  
>>>
```

# Gruppi

- Il pattern seguente fa match con alcuni comuni indirizzi email:

- $\backslash w+@\backslash w+\backslash .+\backslash (com|org|net|edu)$

```
>>> pat1 = "\w+@\w+\.\w+(com|org|net|edu)"  
>>> r1 = re.match(pat,"finin@cs.umbc.edu")  
>>> r1.group()  
'finin@cs.umbc.edu'
```

- Potremmo voler fare match con sotto-parti del pattern, come il nome email e l'host

# Gruppi

- ESEMPI:  
match.py

- E' sufficiente racchiudere tra parentesi i "gruppi" che vogliamo identificare

```
>>> pat2 = "(\w+@\((\w+\.)+(\com|\org|\net|\edu))\"  
>>> r2 = re.match(pat2,"finin@cs.umbc.edu")  
>>> r2.group(1)  
'finin'  
>>> r2.group(2)  
'cs.umbc.edu'  
>>> r2.groups()  
r2.groups()  
('finin', 'cs.umbc.edu', 'umbc.', 'edu')
```

# Gruppi

▪ ESEMPI:  
namedgroup.py

- Notare che i gruppi sono numerati in pre-ordine (cioè rispetto alla loro parentesi di apertura)
- È anche possibile etichettare i gruppi e referenziarli attraverso le etichette

```
>>> pat3 = "(?P<name>\w+@\(?P<host>(\w+\.)+(com|org|net|edu))"
```

```
>>> r3 = re.match(pat3,"finin@cs.umbc.edu")
```

```
>>> r3.group('name')
```

```
'finin'
```

```
>>> r3.group('host')
```

```
'cs.umbc.edu'
```

# Cos'è un match object?

- Il match object è un'istanza della classe `match` con i dettagli del risultato del match
- `>>> pat = "a*b"`
- `>>> r1 = re.search(pat,"fooaaaabcde")`
- `>>> r1.group() # restituisce la stringa che ha fatto match`
- `'aaab'`
- `>>> r1.start() # indice dell'inizio del match`
- `3`
- `>>> r1.end() # indice della fine del match`
- `7`
- `>>> r1.span() # tupla (start, end)`
- `(3, 7)`

# re match object: Alcuni metodi utili

- **group([group1, ...])**
  - Restituisce le occorrenze dei sottopattern(gruppi)
- **start([group])**
  - Restituisce la posizione di inizio dell'occorrenza di un dato gruppo
- **end([group])**
  - Restituisce la posizione di fine dell'occorrenza di un dato gruppo
- **span([group])**
  - Restituisce le posizioni di inizio e fine

# re.split()

- ESEMPI:  
resplit.py

- **re.split()** è simile a **split()** ma può utilizzare dei pattern

```
>>> some_text = 'alpha, beta,,,gamma delta'  
>>> re.split('[, ]+', some_text)  
['alpha', 'beta', 'gamma', 'delta']
```

- Con il parametro **maxsplit** è anche possibile indicare il massimo consentito numero di suddivisioni

```
>>> re.split('[, ]+', some_text, maxsplit=2)  
['alpha', 'beta', 'gamma delta']  
>>> re.split('[, ]+', some_text, maxsplit=1)  
['alpha', 'beta,,,gamma delta']
```

# Altre funzioni del modulo re

- **re.findall()** trova tutti i match:

```
>>> re.findall("\d+","12 dogs,11 cats, 1 egg")  
['12', '11', '1']
```

- **re.sub()** sostituisce un pattern con una stringa specificata

```
>>> re.sub('(blue|white|red)', 'black', 'blue socks and  
red shoes')  
'black socks and black shoes'
```

# sub e i gruppi

- È possibile compiere sostituzioni avanzate utilizzando `re.sub()` insieme a riferimenti a gruppi
- Ad esempio, immaginiamo di voler sostituire `*something*` con '`<em>something</em>`':
  - `>>> emphasis_pattern = r'\*([^\*]+\*)\*'`
  - `>>> re.sub(emphasis_pattern, r'<em>\1</em>', 'Hello, *world*!')`
  - `'Hello, <em>world</em>!'`

# sub e le funzioni di sostituzione

- È possibile compiere sostituzioni avanzate utilizzando vere e proprie funzioni di sostituzione invece che una semplice stringa

```
def dashrepl(matchobj):  
    if matchobj.group(0) == '-':  
        return ''  
    else:  
        return '-'
```

```
>>>re.sub('{1,2}', dashrepl, 'pro----gram-files')  
'pro--gram files'
```

# Pattern greedy e non greedy

- **ATTENZIONE:** gli operatori di ripetizione sono greedy (ingordi) di default, cioè tentano di trovare i match più grandi possibili
- Questo può a volte produrre risultati non desiderati

```
>>> emphasis_pattern = r'\*(.+)\*'  
>>> re.sub(emphasis_pattern, r'<em>\1</em>',  
'*This* is *it*!')  
'<em>This* is *it</em>!'
```

# Pattern greedy e non greedy

- Per risolvere il problema, è sufficiente utilizzare le versioni non-greedy degli operatori di ripetizione, ad es:
  - + operatore greedy
  - +? operatore non-greedy

```
>>> emphasis_pattern = r'\*(.+?)\*'  
>>> re.sub(emphasis_pattern, r'<em>\1</em>', '*This* is  
*it*!')  
'<em>This</em> is <em>it</em>!'
```

# Compilazione di espressioni regolari

- Se si prevede di utilizzare un pattern re più di una volta, è consigliabile compilarlo, ottenendo un oggetto pattern
- Python produce in questo caso una speciale struttura dati che ne velocizza il matching

```
>>> cpat = re.compile(pat)
>>> cpat
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r = cpat.search("finin@cs.umbc.edu")
>>> r
<_sre.SRE_Match object at 0x895a0>
>>> r.group()
'finin@cs.umbc.edu'
```

# Pattern object: metodi

- Per un pattern object sono definiti metodi analoghi a quelli visti nel modulo re:
  - **match**
  - **search**
  - **split**
  - **findall**
  - **sub**
  -

# Generatori

▪ ESEMPI:  
generators.py

- **Un generatore è una funzione che produce una sequenza di risultati invece di un singolo valore**
- **I valori della sequenza sono ritornati uno alla volta, tramite l'istruzione yield**

```
def countdown(n):  
    while n > 0:  
        yield n  
        n-=1  
>>> for i in countdown(5):  
...     print i,  
...  
5 4 3 2 1  
>>>
```

# Generatori

- Il comportamento di un generatore è diverso da quello di una normale funzione
- L'invocazione di un generatore produce un oggetto generatore
- **NON esegue la funzione!**

```
def countdown(n):  
    print "Counting down from", n
```

```
    while n > 0:
```

```
        yield n
```

```
        n-=1
```

```
>>> x = countdown(10) —
```

```
>>> x
```

```
<generator object at "hex. Address">
```

```
>>>
```

Notate come non  
sia stampato alcun valore

# Generatori

- La funzione viene eseguita invocando il metodo `next()` dell'oggetto generatore
  - Un po' come avviene per gli iteratori

```
>>> x = countdown(10)
```

```
>>> x
```

```
<generator object at “hex. Address”>
```

```
>>> x.next()
```

```
Counting down from 10
```

```
>>>
```

La funzione inizia la sua esecuzione qui

- `yield` produce un valore e sospende la funzione fino alla prossima `next()`

```
>>> x.next()
```

```
9
```

```
>>> x.next()
```

```
8
```

# Generatori

- **Quanto la funzione ritorna, l'iterazione finisce**

```
>>> x.next()
```

```
1
```

```
>>> x.next()
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
```

```
StopIteration
```

La funzione inizia la sua esecuzione qui

# Generatori vs. Iteratori

- **Un oggetto generatore è diverso da un oggetto iteratore**
- **Il generatore è il classico esempio di one-time operation**
  - **L'iterazione avviene una sola volta**
  - **Se si vuole ripetere l'iterazione, è necessario creare un altro generatore**
  - **Un iteratore può iterare su una lista quante volte vuole**
- **Il generatore è più semplice da usare**
  - **Non occorre creare i metodi `.next()`, `__iter__()`, etc.**

# Espressioni generatrici

▪ ESEMPI:  
genlog.py

- È possibile creare sequenze in maniera simile alle list comprehension
- List comprehension: lista creata tramite un ciclo for

```
>>> fruit = [ 'apple ', ' orange ', ' pear ']  
>>> list = [ f.strip() for f in fruit]  
['apple', 'orange', 'pear']  
>>>
```

- Esempio di espressione generatrice

```
>>> a = [1, 2, 3, 4]  
>>> b = (2*x for x in a)  
>>> b  
<generator object at “hex address”>  
>>> for i in b: print i,  
2 4 6 8
```