Programming for Biologists

Working with Strings and Regular Expression

Bioinformatics Programming Using Python

Learning Objectives

- Strings are immutable
- String methods
- How to modify strings
- What are raw strings
- Substring matching
- re module
- Greedy vs Non-greedy matching

Working with Strings

Python strings are immutable sequences:

- Immutable: cannot be directly modified in-place
- Sequences: ordered collections that are accessed by offset

Strings support:

- Operations: indexing, slicing, concatenation, etc.
- Functions that operate on collections,
- Methods: functions associated with specific string objects, changing strings, parsing text
- Formatting: using either expressions or methods
- Matching

String Literals

Assignment	Interpretation
S = ''	Empty string
S = "spam's"	Double quotes, same as single
S = 's\np\ta\x00m'	Escape sequences
S = """..."""	Triple-quoted block strings
S = r'\temp\spam'	Raw strings

String Operations

Operation	Interpretation
S1 + S2	Concatenate
S * 3	Repeat
S[i]	Index
S[i:j]	Slice
len(S)	Length

```
kind = "green"
"a {} parrot".format(kind) # string formatting
f"a {kind} parrot"           # alternative syntax "f-string"
```

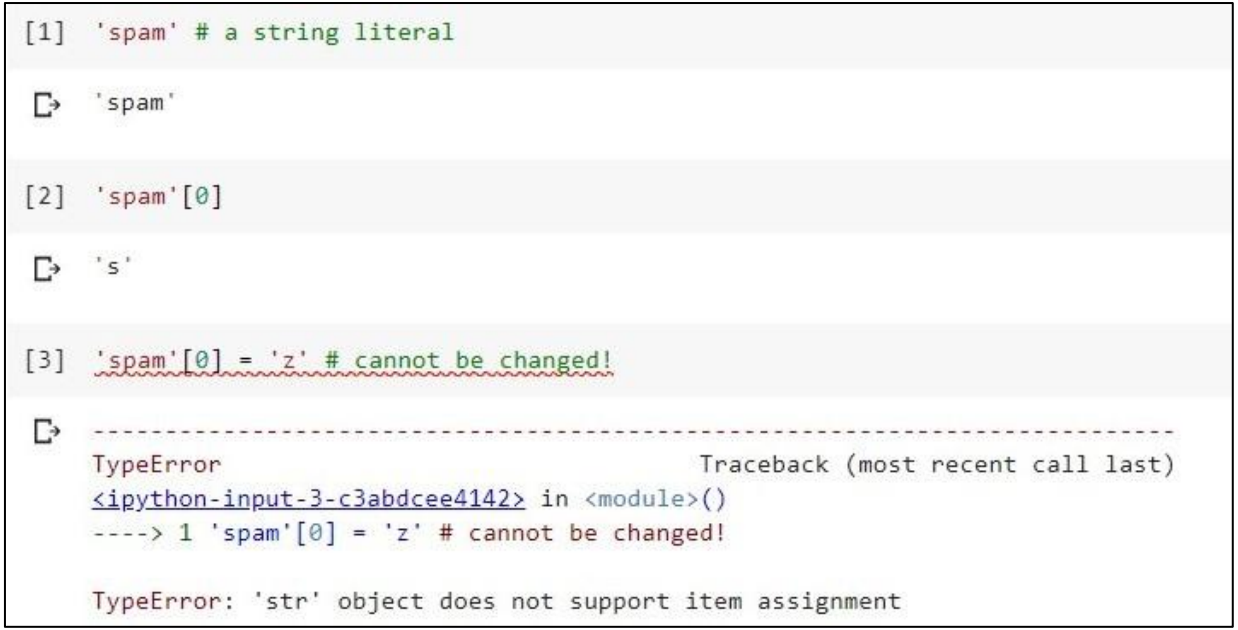
Method	Interpretation
"a {} parrot".format(kind)	String formatting method
S.find('pa')	String method calls: search,
S.rstrip()	String method calls: search,
S.replace('pa', 'xx')	replacement,
S.split(',')	split on delimiter,
S.isdigit()	case conversion,
S.lower()	end test,
'spam'.join(strlist)	delimiter join,
S.encode('latin-1')	Unicode encoding, etc.

String Expressions and Functions

Expression	Interpretation
for x in S: print(x)	Iteration
'spam' in S	membership (test)
[c * 2 for c in S]	list comprehension
map(ord, S)	function call on iterable

Changing Strings

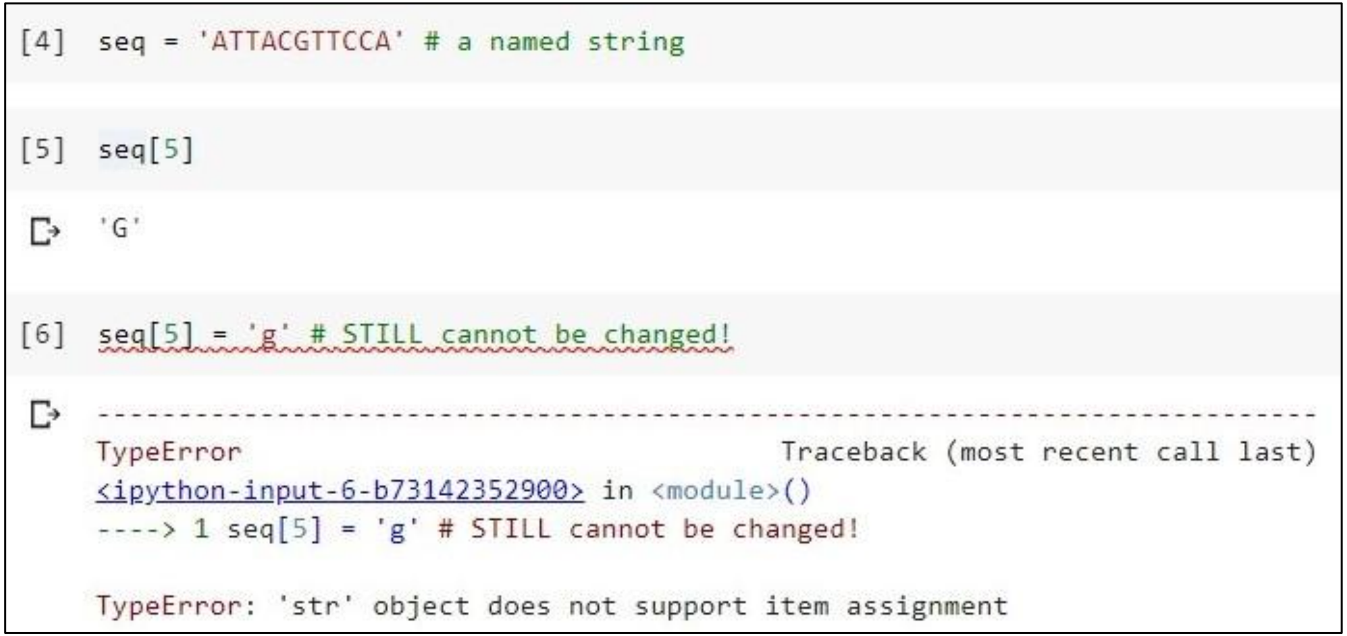
- Because strings are immutable, they cannot be changed directly:



```
[1]  'spam' # a string literal

     'spam'

[2]  'spam'[0]

     's'

[3]  'spam'[0] = 'z' # cannot be changed!

     ---------------------------------------------------------------------------
     TypeError                                 Traceback (most recent call last)
     <ipython-input-3-c3abdcee4142> in <module>()
     ----> 1 'spam'[0] = 'z' # cannot be changed!

     TypeError: 'str' object does not support item assignment
```

Changing Strings

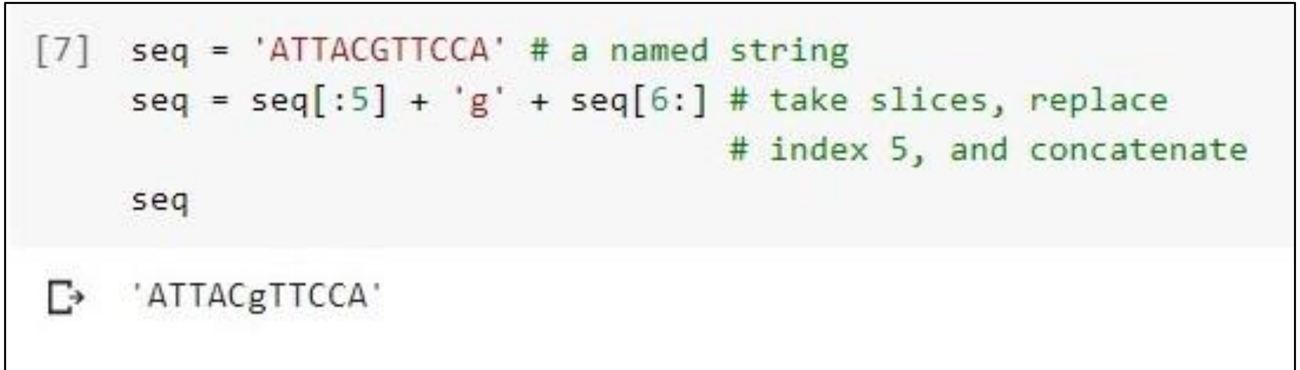
- A string object is an item in memory that is referenced by a name (i.e. a pointer to the object)



```
[4]  seq = 'ATTACGTTCCA' # a named string

[5]  seq[5]

     'G'

[6]  seq[5] = 'g' # STILL cannot be changed!

     ---------------------------------------------------------------------------
     TypeError                                 Traceback (most recent call last)
     <ipython-input-6-b73142352900> in <module>()
     ----> 1 seq[5] = 'g' # STILL cannot be changed!

     TypeError: 'str' object does not support item assignment
```

Changing Strings

- You CAN change the value of the string object using operations, functions, and methods that operate on strings, and these always make a new copy of the string.
- So, to change strings, you must operate on a copy (copies) of the string and reassign the new value to the name that references the string:

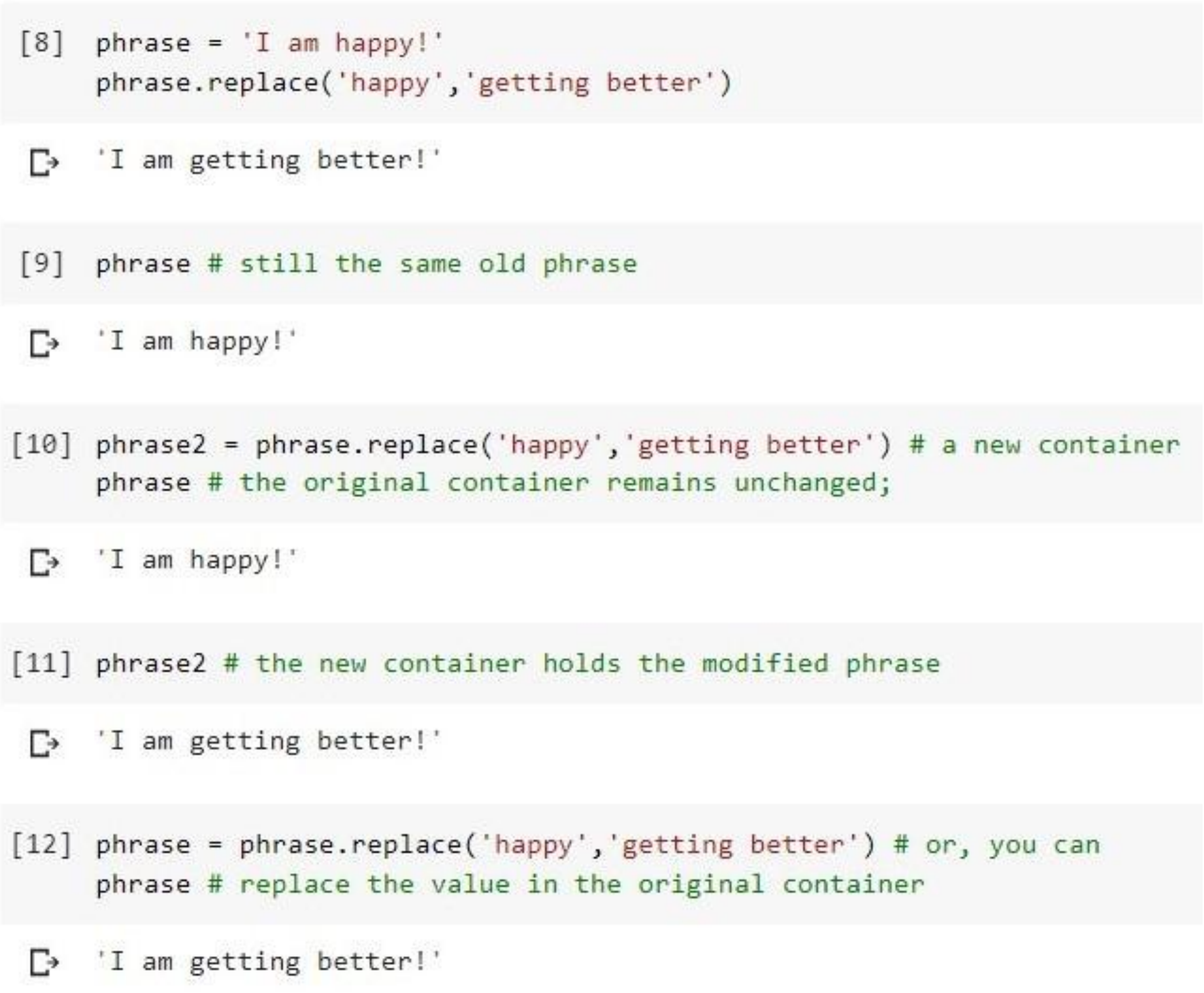


```
[7] seq = 'ATTACGTTCCA' # a named string
    seq = seq[:5] + 'g' + seq[6:] # take slices, replace
                                  # index 5, and concatenate
    seq
```

```
'ATTACgTTCCA'
```

Changing Strings: replace method

- The same rule applies to string methods, which operate on named strings (string objects), yielding a new string value.
- Thus, to retain these new string objects, you must assign them to a name



```
[8]  phrase = 'I am happy!'
     phrase.replace('happy','getting better')
```

```
'I am getting better!'
```

```
[9]  phrase # still the same old phrase
```

```
'I am happy!'
```

```
[10] phrase2 = phrase.replace('happy','getting better') # a new container
     phrase # the original container remains unchanged;
```

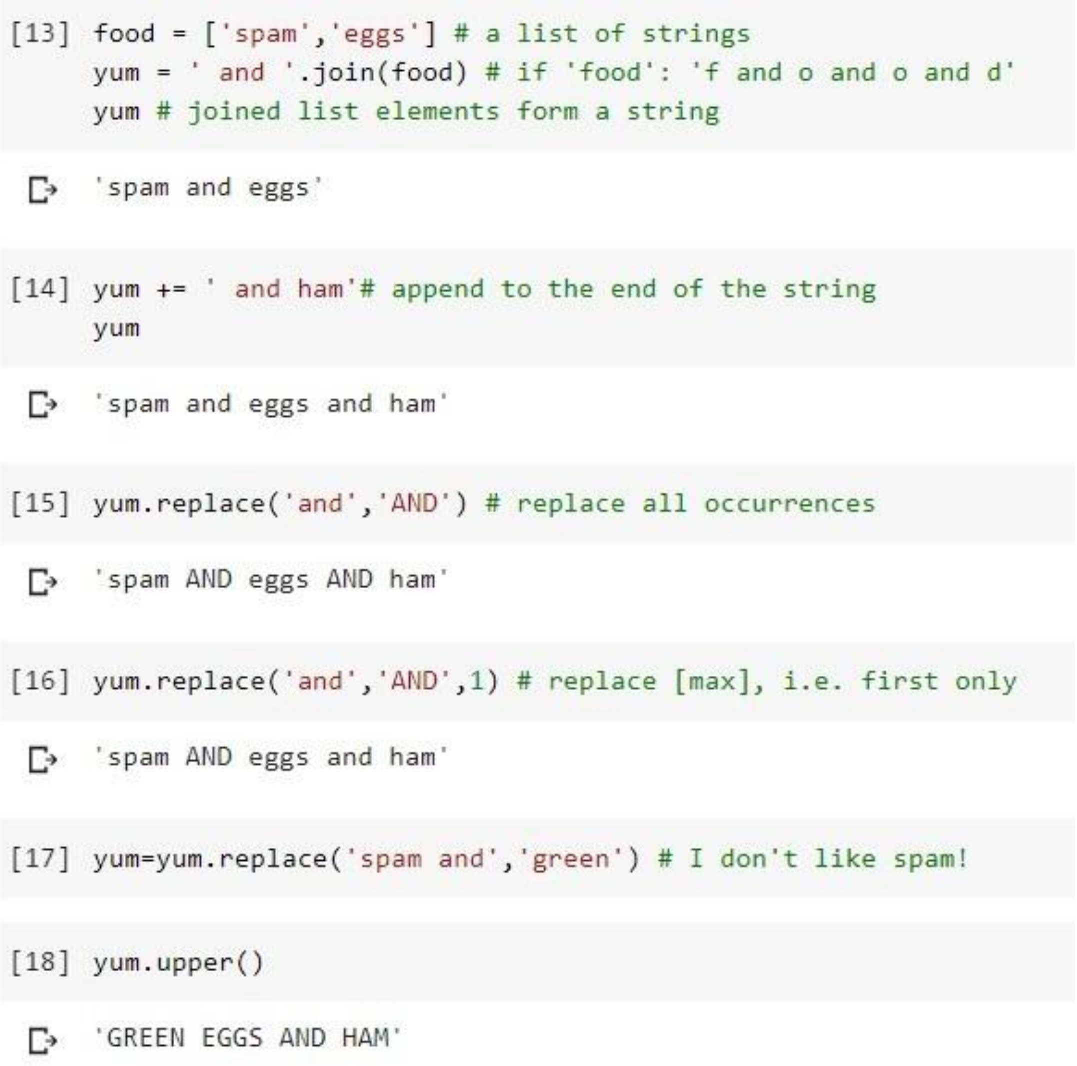
```
'I am happy!'
```

```
[11] phrase2 # the new container holds the modified phrase
```

```
'I am getting better!'
```

```
[12] phrase = phrase.replace('happy','getting better') # or, you can
     phrase # replace the value in the original container
```

```
'I am getting better!'
```



```
[13] food = ['spam','eggs'] # a list of strings
     yum = ' and '.join(food) # if 'food': 'f and o and o and d'
     yum # joined list elements form a string
```

```
'spam and eggs'
```

```
[14] yum += ' and ham'# append to the end of the string
     yum
```

```
'spam and eggs and ham'
```

```
[15] yum.replace('and','AND') # replace all occurrences
```

```
'spam AND eggs AND ham'
```

```
[16] yum.replace('and','AND',1) # replace [max], i.e. first only
```

```
'spam AND eggs and ham'
```

```
[17] yum=yum.replace('spam and','green') # I don't like spam!
```

```
[18] yum.upper()
```

```
'GREEN EGGS AND HAM'
```

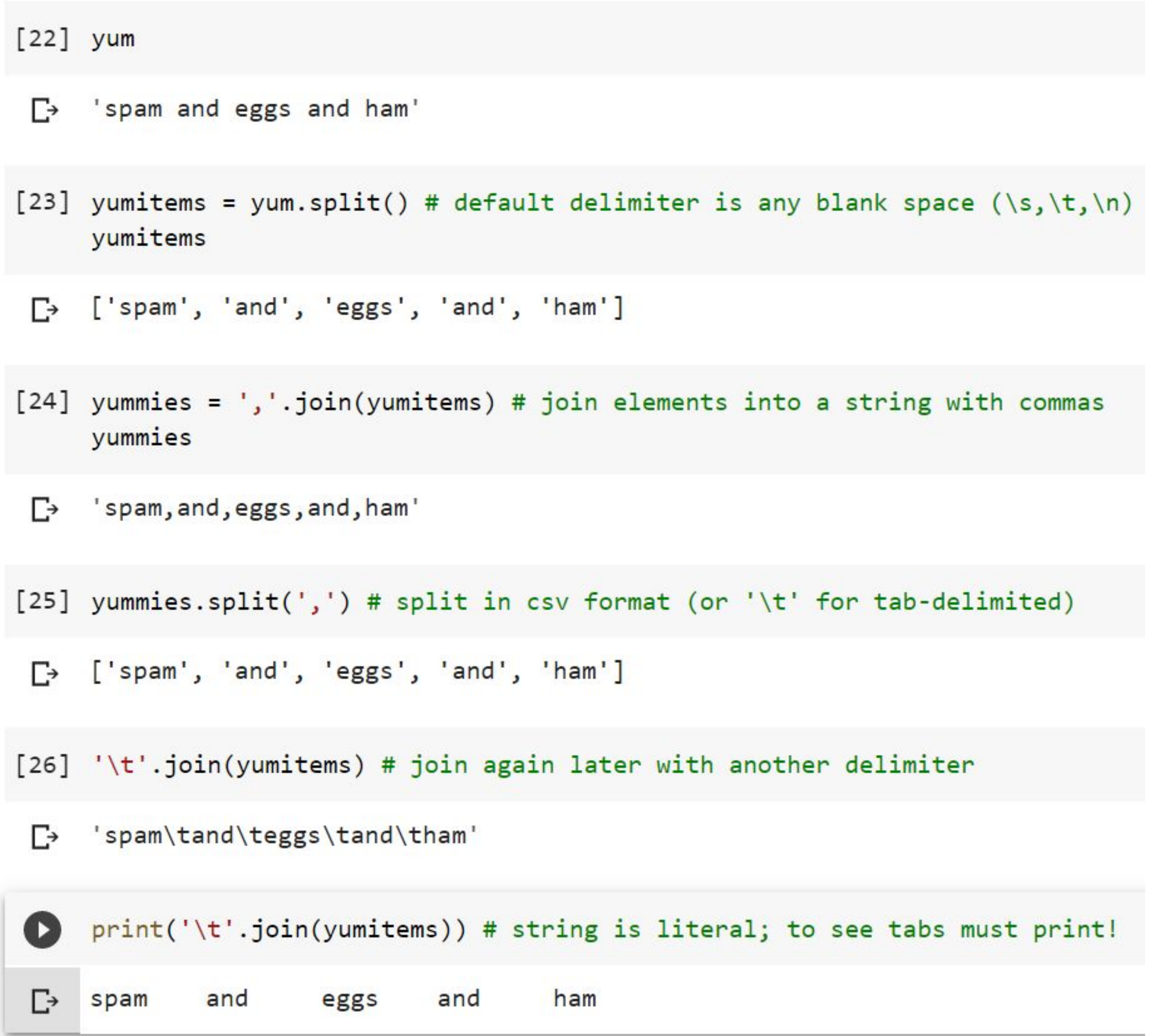
Changing Strings: list conversion

- If you want to apply many changes to a very large string, it might be more efficient to convert your string to a list (which is mutable, and thus supports in-place changes) instead of using string operations or methods (which create copies of strings).
- To replace parts of a string after list conversion, you can use list slices.

```
yum = 'spam and eggs and ham'
yumlist = list(yum) # convert a string to a list
yumlist
['s','p','a','m',' ','a','n','d',' ','e','g','g','s',' ','a','n','d','
','h','a','m']

yumlist[0:7] = list('green ') # want to replace 'spam and ' with 'green '
yumlist
['g','r','e','e','n',' ','d',' ','e','g','g','s',' ','a','n','d','
','h','a','m']

# Oops! Same as yumlist = list('green ') + yumlist[:7], that is:
# We replaced not 8 but 7 elements, i.e. ['s','p','a','m',' ','a','n'],
# with 6 -- and the length of yumlist contracts to 20 from 21.

del yumlist[5:7] # removes [' ','d'] b/c slices are exclusive of 2nd index
yumlist
['g','r','e','e','n',' ','e','g','g','s',' ','a','n','d',' ','h','a','m']

yum = ''.join(yumlist) # rejoin into single string after removing 2 chars
yum
'green eggs and ham'
```

Parsing Text

- You can use slices to extract parts of strings, as in the previous example, or
- You can use the split method to chop the string into individual columns using a specified delimiter string:



```
[22] yum
```

```
'spam and eggs and ham'
```

```
[23] yumitems = yum.split() # default delimiter is any blank space (\s,\t,\n)
     yumitems
```

```
['spam', 'and', 'eggs', 'and', 'ham']
```

```
[24] yummies = ','.join(yumitems) # join elements into a string with commas
     yummies
```

```
'spam,and,eggs,and,ham'
```

```
[25] yummies.split(',') # split in csv format (or '\t' for tab-delimited)
```

```
['spam', 'and', 'eggs', 'and', 'ham']
```

```
[26] '\t'.join(yumitems) # join again later with another delimiter
```

```
'spam\tand\teggs\tand\tham'
```

```
print('\t'.join(yumitems)) # string is literal; to see tabs must print!
```

```
spam    and     eggs    and     ham
```

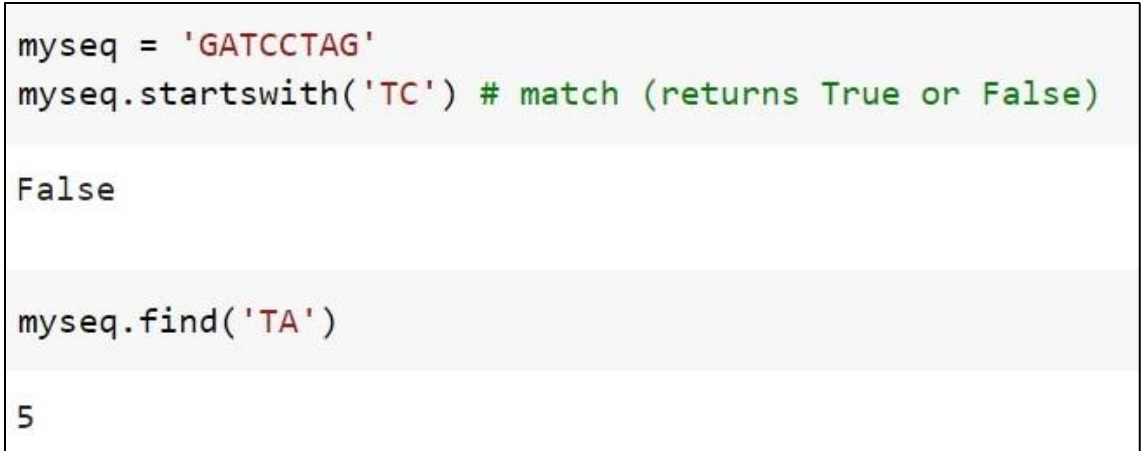
Basic Syntax

- Special characters within regular expression strings are interpreted a little differently than in regular Python strings.
- To avoid confusion, ALWAYS USE RAW STRINGS for regular expressions:
- Regular expressions contain a lot of "funny" characters that need to be interpreted literally; using raw strings ensures that these are interpreted as they are intended.

```
 ---
'\n' # a regular string
'\n'

len('\n') # the symbol '\n' is a single character
1

r'\n' # the raw string '\n' is not "escaped"
'\\n'

len(r'\n') # so the backslash is interpreted literally
2
```

Simple Fixed-length Substring Matching

- Note: Python distinguishes between matching and searching, which other languages do not.
  - **match** = looks for a pattern at the start of the target
  - **search** = looks for pattern anywhere in the target
- We will usually use 'match' to refer to both cases.



```
myseq = 'GATCCTAG'
myseq.startswith('TC') # match (returns True or False)
```

```
False
```

```
myseq.find('TA')
```

```
5
```

Simple Fixed-length Substring Matching

- To find multiple substrings in a sequence, you could define a function:

```
def multi_search(target, patterns):
    """Return the first position in target where any
of the strings in patterns list is found"""
    return min([target.find(pattern) \
        for pattern in patterns \
            if target.find(pattern) >= 0])

multi_search(myseq, ('TA', 'TC')) # returns 2
```

Character Sets and Classes

Pattern	Matches
[ACTG]	One DNA base character
[A-Za-z_]	One underscore or letter
[^0-9]	Any character except a digit
[-+/*^]	Any of +, -, /, *, ^; ^ does not negate the others because it is not the first character in the set
[0-9\t]	A tab or a digit
.	Any character

Character Sets and Classes

Character	Matches
`\d`	Any digit
`\D`	Any non-digit
`\s`	Any whitespace character
`\S`	Any non-whitespace character
`\w`	Any character considered part of a word
`\W`	Any character not considered part of a word

Boundaries

Special notation to indicate the beginning or end of a target, a line, or a word:

Character	Matches
^	The start of a line or the beginning of the pattern
$	The end of a line or the end of the pattern
\A	The start of the pattern only
\Z	The end of the pattern only
\b	The boundary between a word and nonword character or vice versa
\B	Anywhere except the boundary between a word and nonword character or vice versa

Examples

pattern	Matches
g..t	"gaat", "goat", "gotta get a goat" (twice)
g[gatc][gatc]t	"gaat", "gttt", "gotta get an aggat" (once)
\d\d\d-\d\d\d\d	998-8200 and 212-998-8200, but not 011-21-299-88-2000
^\d\d\d-\d\d\d\d	998-8200 and 212-9988200, but not 212-998-8200
^\d\d\d-\d\d\d\d$	ONLY 7-digit telephone numbers with a dash (e.g. 998-8200)
\bcat	"cat", "catty", and "more catsup please", but not "scatter"
\bcat\b	text containing the word "cat"

Quantifiers

By default, an atom matches once. This can be modified by following the atom with a quantifier:

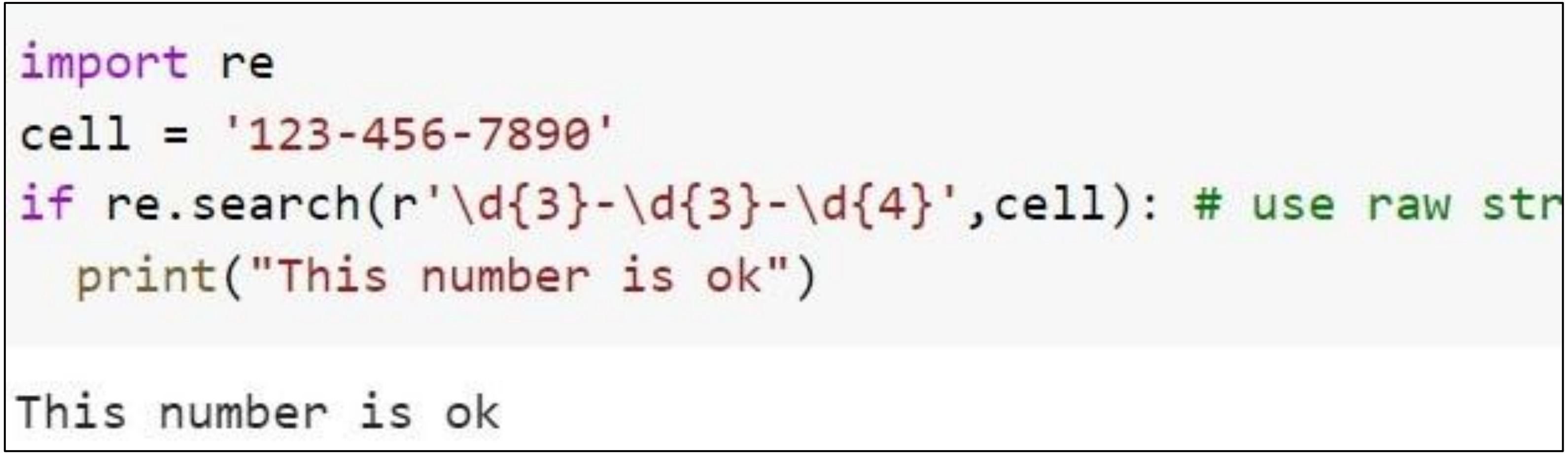
Character	Matches
?	Atom matches zero times or exactly once
*	Atom matches zero or more times
+	Atom matches one or more times
{n}	Atom matches exactly n times
{m,n}	Atom matches between m and n times, inclusive
{m,}	Atom matches at least m times
{,n}	Atom matches at most n times

Examples

pattern	Matches
goa?t	"got", "goat", "gotta get a goat" (twice)
g.+t	"get", "got", "goat", "grant", etc... and "gotta get a great goat" (4x)
g.*t	"gt", "get", "goat", "greet", "grandest", etc.
^\d{3}-\d{4}$	only matches 7-digit telephone numbers with a dash (e.g. 998-8200)

The re module

- The Python re module provides functions and methods that use regular expressions for a variety of actions. For example:

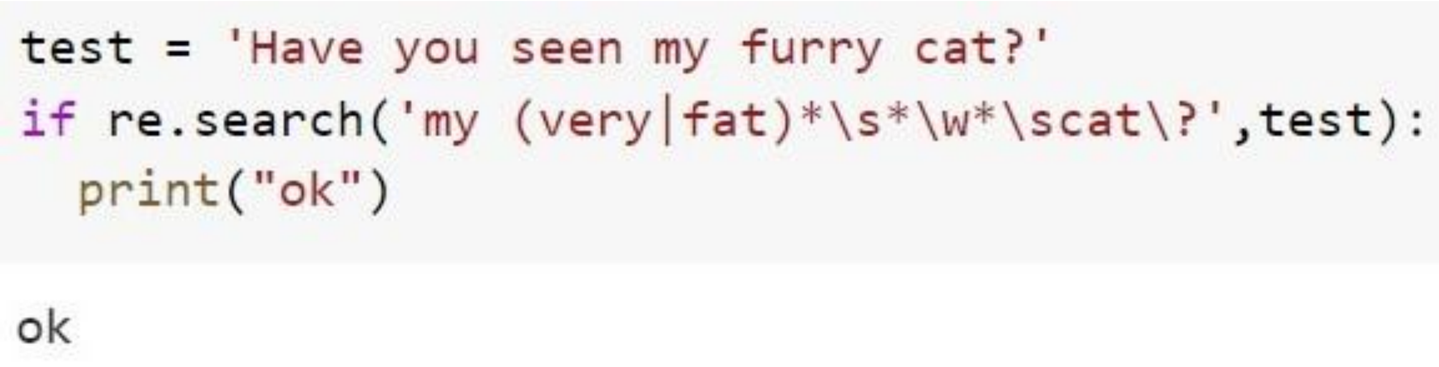


```
import re
cell = '123-456-7890'
if re.search(r'\d{3}-\d{3}-\d{4}',cell): # use raw str
  print("This number is ok")
```

```
This number is ok
```

Grouping and Alternation

- Enclosing more than one character, or parts of a regular expression, within parentheses will cause quantifiers to apply to the entire group.
- You can search for alternatives within parentheses using the '|' symbol:



```
test = 'Have you seen my furry cat?'
if re.search('my (very|fat)*\s*\w*\scat\?',test):
  print("ok")
```

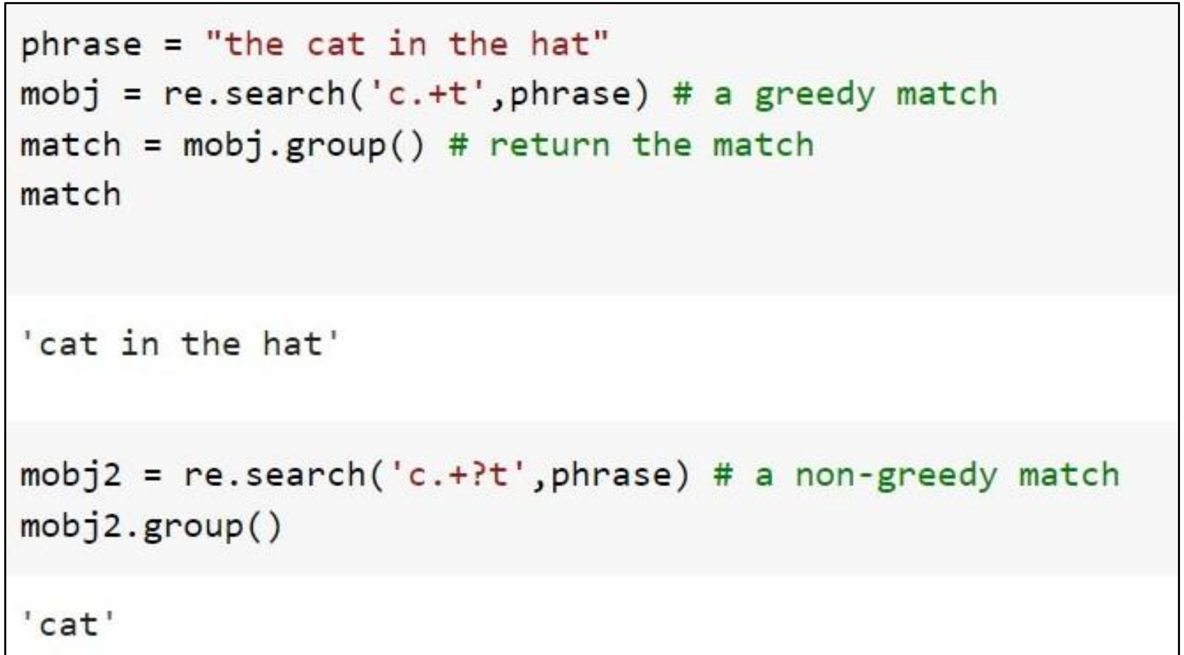
```
ok
```

Match objects

- So far we've learned how to test for matching, but we don't know how to capture the results of the match.
- Many of the results of the re module's methods and functions return match objects.
  - Match objects capture the parts of each match in a target string that correspond to one of the pattern's parenthesized groups.
- Match objects allow you to capture not only the contents of each match, but also extra bits of information associated with the match:
  - The groups captured in match objects are an analysis of the structure of the matched string.
  - Match object fields can be accessed using the dot notation matchobj.field: startpos, endpos, re, string, lastindex

Example with match objects

- We can use the group method on the matched object to retrieve the contents of the match:



```
phrase = "the cat in the hat"
mobj = re.search('c.+t',phrase) # a greedy match
match = mobj.group() # return the match
match
```

```
'cat in the hat'
```

```
mobj2 = re.search('c.+?t',phrase) # a non-greedy match
mobj2.group()
```

```
'cat'
```

Match objects: re.compile

- The re.compile function returns a regular expression object that encapsulates a data structure built from the pattern.

- Using a compiled regular expression object can be more efficient when the regular expression is used more than once (since it is compiled only once).