More on Lists(Strings) and List(String) Comprehensions

Lecture 04.05

By Marina Barsky

Lists (strings) are Python ***objects***

They have attached *functions* that

work on the object itself

These functions are called *methods*

```
t = [1, 2, 3, 2]
t.count(2)
i=t.count(3)
```

```
s = 'abc'
s = s.upper()
n = s.count('a')
```

What methods are available?

```
s='abc'
dir(s)
dir(str)
[…'capitalize', 'count', 'encode', 'endswith',
'find', 'format', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'islower',
'isnumeric', 'isprintable', 'isspace',
'isupper', 'join', 'lower’, …]
```

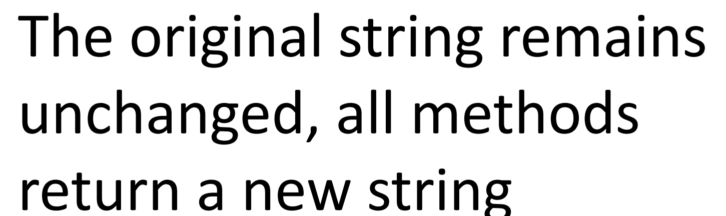
How to use a method?

- **help**(str.find)

```
Help on method_descriptor:
find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

Try str methods in Python shell: 1/2

```
>>> white_rabbit = "I'm late! I'm late! For a very important date!"

>>> white_rabbit.lower()
>>> white_rabbit.find('late')
>>> white_rabbit.count('ate')
>>> white_rabbit.find('late',7)
>>> white_rabbit.find('Late')
>>> white_rabbit.rfind('late')
>>> white_rabbit
```



Try str methods in Python shell: 2/2

```
>>> "computer".capitalize()

>>> s="      I'm feeling spaced out.        "
>>> s.rstrip()
>>> s.strip()

>>> robot = 'R2D2'
>>> robot.isupper()
>>> robot.isalpha()
>>> robot.isdigit()
>>> robot.isalnum()
```

List methods

```
>>> dir(list)
[…,'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Methods that modify the list

Method	Description	Example
list.**append**(obj)	Append object to the end of list.	`>>> colors = ['yellow', 'blue']` `>>> colors.append('red')`
list.**extend**(list1)	Append the items in the list1 parameter to the list.	`>>> colors.extend(['pink', 'green'])`
list.**pop**([index])	Remove the item at the end of the list; optional index to remove from anywhere.	`>>> colours.pop()` `>>> colours.pop(2) 'red'`

list.**remove**(obj)	Remove the first occurrence of the object; error if not there.	`>>> colours.remove('green’)`
		`Traceback (most recent call last): … ValueError: list.remove(x): x not in list`
		`>>> colours.remove('pink')`

list.**reverse**()	Reverse the list.	`>>> grades = [95, 65, 75, 85]` `>>> grades.reverse()`
list.**sort**()	Sort the list from smallest to largest.	`>>> grades.sort()`
list.**insert**(int, obj)	Insert object at the given index, moving items to make room.	`>>> grades.insert(2, 80)`

Methods that obtain information (read) form the list

Method	Description	Example
list.**count**(object)	Return the number of times object occurs in list.	`>>> letters = ['a', 'a', 'b', 'c’]` `>>> letters.count('a') 2`

list.**index**(object)	Return the index of the first occurrence of object; error if not there.	`>>> letters.index('a’)`
		`0`
		`>>> letters.index('d')`
		`Traceback (most recent call last): …`
		`ValueError: 'd' is not in list`

List mutability

- We say that lists are ***mutable***: they can be modified
- All the other types we have seen so far (range, str, int, float and bool) are ***immutable***: they cannot be modified

```
>>> classes = ['chem', 'bio', 'cs', 'eng']
>>> # Elements can be added:
>>> classes.append('math')
['chem', 'bio', 'cs', 'eng', 'math']
>>> # Elements can be replaced:
>>> classes[1] = 'soc'
['chem', 'soc', 'cs', 'eng', 'math']
>>> # Elements can be removed:
>>> classes.pop()
['chem', 'soc', 'cs', 'eng']
```

Aliasing mutable variables

```
lst1 = [0, 2, 4, 6]
lst1[2] = 5

lst2 = lst1
lst1[-1] = 17

print(lst1)
print(lst2)
```

- We modified lst2 through lst1, because they both point to the same memory address
- lst2 is not a new list, but it is an ***alias*** of lst1

Immutable parameters cannot be modified inside the function

```
def conform(fav):
""" sets any number to my favorite number 42 """
    fav = 42

fav = 7
conform(fav)
print(fav)
```

- *int* is immutable – we cannot change value pointed to by fav – we can just change it to point to a new int
- The original variable remains unchanged

Mutable parameters can be modified inside the function

```
def double_first(t):
    """ doubles element 0 of t
    """
    t [0] = t [0] * 2

lst = [40, 30, 50]
print(lst)
double_first(lst)
print(lst)
```

Mutable and immutable parameters

- When mutable objects are passed to a function, a new alias reference to this object is created – both refer to the same original object
- Because both original and copy refer to the same place in memory, and the content of mutable objects can be modified - by changing content from a copy, we affect the original object

Immutable:	**Mutable**:
int	*list*
float	*dictionary*
str	*set*
range	…

Safe programming with mutables

- Passing immutable objects is safe: they cannot be accidentally modified inside the function
- Passing mutable objects is unsafe: they can be modified inside the function
- It is safer to create a new mutable object inside the function and **return** it instead

- Example:

`t.sort()` ← changes the content of list t

`sorted_t = sorted(t)` ← puts sorted t into a new variable, `t` itself remains unchanged

Converting between lists and strings

- Each string *s* can be converted into a list using the string method s.*split*(*separator*):

```
'one,two,three'.split(',')
```

- Each **list** *t* **of strings** can be converted into a single string using the list method glue.*join*(t)

```
','.join(['one', 'two', 'three'])
```

The split() method cuts the string into a sequence of variables

```
rock_band = "Al Carl Mike Brian”
(rhythm, lead, vocals, bass) = rock_band.split()
```

This sequential type is called a ***tuple***

We have encountered tuples before. Where?

```
int_seq = range(5)
(0,1,2,3,4)
```

range function also produces a ***tuple***

Tuples Are Like Lists

Tuples are another kind of *iterable* that works much like a list - they have elements which are indexed starting at 0

```
>>> x = ('Glenn', 'Sally', 'Joseph')
>>> print(x[2])
Joseph
>>> y = ( 1, 9, 2 )
>>> print(y)
(1, 9, 2)
>>> print(max(y))
9
```

```
>>> for iter in y:
...     print(iter)
...
1
9
2
>>>
```

but... Tuples are “immutable”

Unlike a list, once you create a **tuple**, you **cannot alter** its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
>>>[9, 8, 6]
>>>
```

```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback:'str'
object does
not support item
Assignment
>>>
```

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple'
object does
not support item
Assignment
>>>
```

Things not to do With Tuples

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no attribute 'reverse'
>>>
```

Tuple vs. lists

```
>>> lst = list()
>>> dir(lst)
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']

>>> t = tuple()
>>> dir(t)
['count', 'index']
```

Tuples are More Efficient

Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists

So in our program when we are making “temporary variables” we prefer tuples over lists

Tuples as variables

We can also put a tuple on the left-hand side of an assignment statement

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred
>>> (a, b) = (99, 98)
>>> print(a)
99
```

LIST COMPREHENSION

List comprehension is a transformation applied to each element of the list (string, range)

The result of this operation is a new list with transformed elements

Example: discount

- Problem: apply a 20% discount to the list of prices.

input: list of old prices

output: list of new prices

- Looping:

```
def apply_discount (t, discount):
    r = []
    for x in t:
        r += [x*discount]

    return r

print(apply_discount ([10,20,30,100], 0.8))
```

Discount with one line of code

- Problem: apply a 20% discount to the list of prices.

input: list of old prices

output: list of new prices

- With list comprehension:

```
print([x*0.8 for x in [10,20,30,100]])
[8.0, 16.0, 24.0, 80.0]
```

Entering list comprehensions

- ***List comprehension*** is a simultaneous transformation of all elements of a sequence (list or string or tuple)
- We attach **the same transformation** to each element, and we generate a new sequence where each element is a result of this atomic transformation
- Why? The transformations run in parallel and the code is faster

Applying same operation to each element of the list

```
[2*x for x in t]
```

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

```
2*(1)   2*(2)   2*(3)   2*(4)   2*(5)
```

```
[2, 4, 6, 8, 10]
```

What is this code doing?

```
>>> [ 2*x for x in [0,1,2,3,4,5] ]
[0, 2, 4, 6, 8, 10]
```

```
>>> [ y**2 for y in range(6) ]
[0, 1, 4, 9, 16, 25]
```

```
>>> [ c == 'a' for c in 'go away!' ]
[False, False, False, True, False,
True, False, False]
```

Elements of syntax

Any operation you want to apply to each element of the list

variable that takes on the value of each element

any name is OK!

iterable (list, string, range)

```
>>> [ 2*x for x in [0,1,2,3,4,5] ]
[0, 2, 4, 6, 8, 10]

>>> [ y**2 for y in range(6) ]
[0, 1, 4, 9, 16, 25]

>>> [ c == 'a' for c in 'go away!' ]
[False, False, False, True, False, True,
False, False]
```

List comprehension

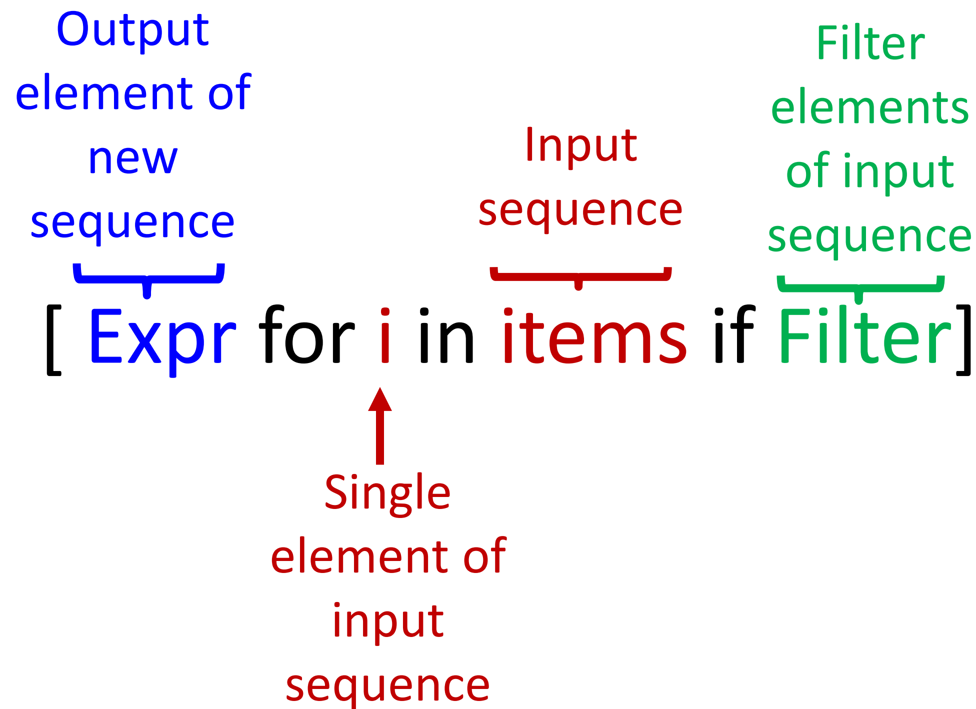
Single element of new output sequence

Input sequence

[Expr for i in items]

Single element of input sequence

List comprehension: with filter



What is printed?

[Expr for i in items if Filter]

```
a_list = [1, 2,'abc, 2.15, 3, 4]

out_list = [i**2 for i in a_list if type(i)==int]

print (out_list)
```

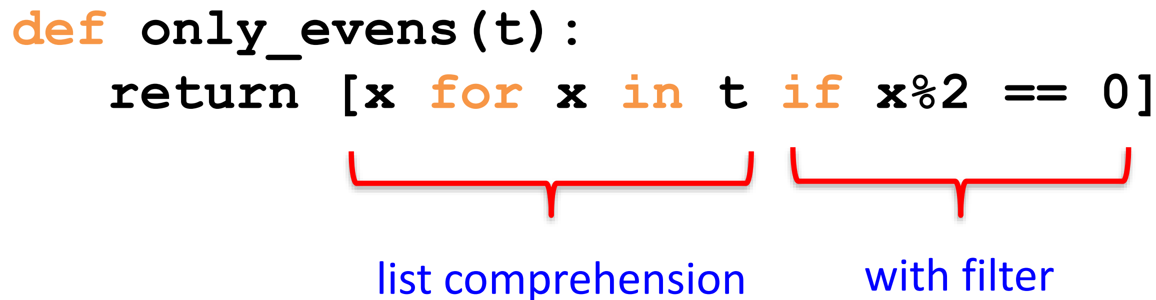
[1, 4, 9, 16]

Loops vs. list comprehensions
`count_vows(s)` # of vowels

```
def count_vows(s):
    count = 0
    for c in s:
        if c in 'aeiou':
            count += 1
    return count
```

```
def count_vows(s):

    return sum([1 for x in s if x in 'aeiou'])
```

Filtering for even numbers




```
>>>only_evens([13, 2, 5, 6, 9])
[2, 6]
```

```
def only_evens(t):
    return [x for x in t if is_even(x)]
```

List comprehension: conditionals

Output element of new sequence if Cond

Output element of new sequence if NOT Cond

Input sequence

[Expr1 if Cond(i) else Expr2 for i in items]

Single element of input sequence

Conditionals: example

[Expr1 if Cond(i) else Expr2 for i in items]

```
>>> lst = [0,3,-1,-4,2]
>>> [2*x if x>0 else -2*x for x in lst]
[0, 6, 2, 8, 4]
```

Breaking list into pairs

```
t = [1,2,3,4,5,6]

pairs = [t[x:x+2] for x in range(0, len(t), 2)]

print (pairs)
```

Examples

Generate all powers of 2 from 0 to 10

```
lst = [2**i for i in range (10)]
# [1 ,2 ,4 ,8 ,16 ,...2^9]
```

Given a list, get a list of square roots of its elements

```
from math import sqrt
lst = [sqrt (x) for x in otherlist ]
# produced a squared list
```

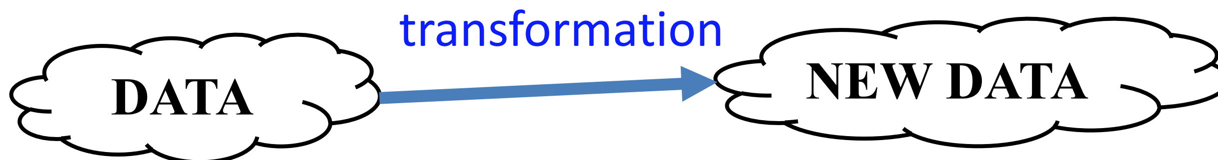
Generate a list of odd numbers from 0 to 10

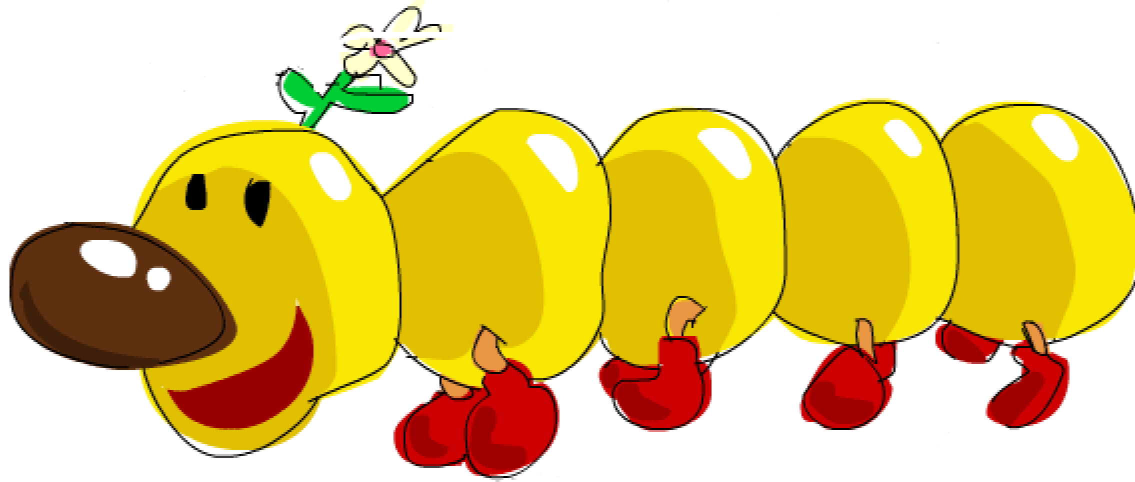
```
list=[x for x in range(10) if x % 2 == 1]
# [1, 3, 5, 7, 9]
```

Why list comprehensions?

- List Comprehensions are at least 35% faster than FOR loop
- They apply transformations to each element of the list in parallel (say, using multiple cores)
- They are a syntax shortcut for more general concept of *mapping*

- The type of computation when data is transformed into the output without intermediate states is the basis of *functional programming*





[*butterfly*(e) for e in caterpillar]