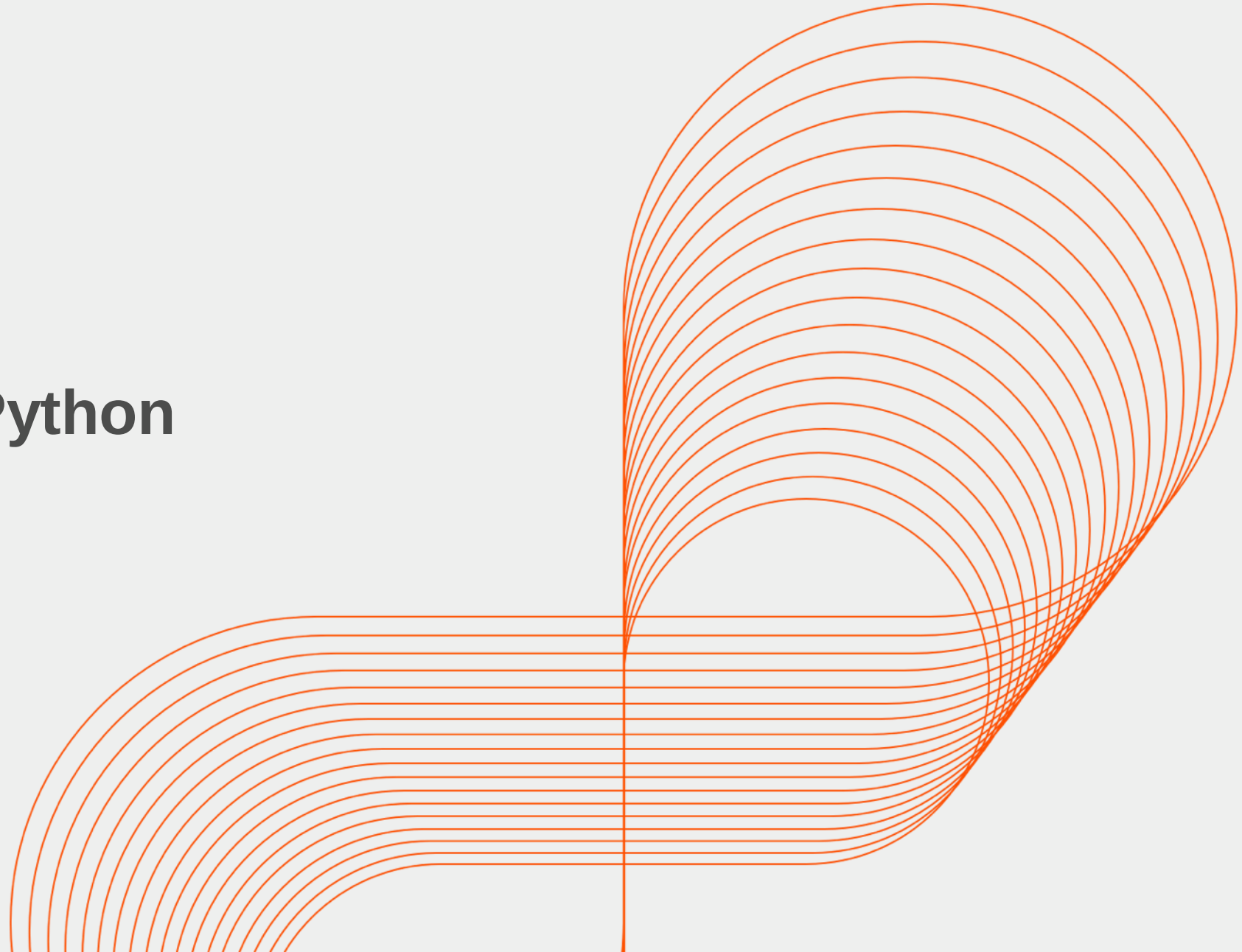
Persistent

Functions in Python



Objectives

At the end of this module, you will be able to:

- Get an overview of Python Functions
- Default Argument Values
- Keyword & Positional Arguments
- Returning Values
- Arbitrary Argument Lists
- Lambda Expressions
- Lambda Forms
- List comprehension
- Map and filter function

Persistent

Functions

What are functions?

- Functions are the structured or procedural programming way of organizing the logic in programs
- A function is a device that groups a set of statements so they can be run more than once in a program
- Functions are also the most basic program structure that Python provides for maximizing code reuse
- Large blocks of code can be neatly segregated into manageable chunks, and space is saved by putting of-repeated code in functions as opposed to multiple copies everywhere
- This also helps with consistency because changing the single copy avoids the need to search for and make changes to multiple copies of duplicated code
- A Python function is a code block. Code blocks in Python are identified by indentation rather than using symbols like curly braces

Function return values and function types

- Functions may return a value back to its caller or do not explicitly return anything at all (None) if they are more procedural in nature
- The following function acts as a procedure returning no value or None as shown below

```
>>> def hello():
...         print 'hello world‘
>>> result=hello()
hello world
>>> result
None
```


Persistent

Function return values and function types (contd.)

- Like most other languages, Python functions return only one value/object.
- The following example shows a function func1 returning a list and another function func2 return a tuple. Because of the tuple's syntax of not requiring the enclosing parentheses, it creates an illusion of returning multiple items.

```
def func1():
    return ['xyz', 10, -8.6]
def func2():
    return ‘xyz', [30, 'python‘], “hello“
```

Default values for arguments

- You can provide default values for a function's arguments
- These arguments are optional when the function is called

```
>>> def myfun(b, c=3, d="hello"):
        return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8

Persistent

Keyword arguments

- You can call a function with some or all of its arguments out of order as long as you specify their names
- You can also just use keywords for a final subset of the arguments

```
>>> def myfun(a, b, c):
        return a-b
>>> myfun(2, 1, 43)
  1
>>> myfun(c=43, b=1, a=2)
  1
>>> myfun(2, c=43, b=1)
  1
```


Persistent

Variable number of arguments

- Python functions can also be defined to handle variable numbers of arguments
- This can be done in two different ways:
  - One way handles the relatively familiar case where an unknown number of arguments are collected at the end of the argument list into a tuple
  - The other method can collect an arbitrary number of keyword-passed arguments, which have no correspondingly named parameter in the function parameter list, into a dictionary
- These two mechanisms are discussed next

Variable number of positional arguments

Example:

```
>>> def tupleVarArgs(arg1, arg2='defaultB', *theRest):
        print ('formal arg 1:', arg1)
        print ('formal arg 2:', arg2)
        for eachXtrArg in theRest:
            print ('another arg:', eachXtrArg)
>>> tupleVarArgs('abc', 123, 'xyz', 456.789)
formal arg 1: abc
formal arg 2: 123
another arg: xyz
another arg: 456.789
```

Variable number of keyword arguments

Example:

```
>>> def dictVarArgs(arg1, arg2='defaultB',
**theRest):
        print ('formal arg 1:', arg1)
        print ('formal arg 2:', arg2)
        for eachXtrArg in theRest.keys():
            print('Xtra arg %s: %s' % \
            (eachXtrArg,
            str(theRest[eachXtrArg])))
```

```
>>> dictVarArgs(1220, 740.0, c='grail')
        formal arg 1:  1220
        formal arg 2:  740.0
        Xtra arg c: grail
```


Persistent

Local and global variables

Example:

```
>>> def fact(n):
...     r = 1
...     while n > 0:
...         r = r * n
...         n = n - 1
...         return r
```


Persistent

Local and global variables (contd.)

Global variables can be accessed and changed by the function.

Example:

```
>>> def func1():
…       global a
…        a = 1
…        b = 2
>>> a = “one”
>>> b = “two”
>>> func1()
>>> a
1
>>> b
‘two’
```

Anonymous functions using Lambda

- Python allows to create anonymous functions using the Lambda keyword.
- The syntax for anonymous functions using Lambda is as follows

  lambda [arg1[, arg2, … argN]]: expression

  Arguments are optional, and if used, are usually part of the expression as well.

Example:

def true(): return 1 can be rewritten as a lambda function as below

>>> lambda:1

<function <lambda> at 0x029F4E30>

Persistent

Anonymous functions using Lambda (contd.)

- In the above example, a Lambda function was created, but was not saved anywhere nor was called

- To keep the object around, this can be saved it into a variable and invoked any time after as shown below

```
>>> true = lambda :1
>>> true()
1
```

- A Lambda expression works just like a function, creating a frame object when called

Small functions and the lambda expression

- If the function you need doesn’t exist, you need to write it. One way to write small functions is to use the lambda statement. lambda takes a number of parameters and an expression combining these parameters, and creates a small function that returns the value of the expression:
- **Examples:**

```
lowercase = lambda x: x.lower()
```

- print_assign = lambda name, value: name + '=' + str(value)

```
adder = lambda x, y: x+y
```

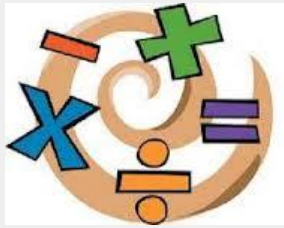
Assignments

1. Accept 2 numbers from keyboard. Pass these as keyworded arguments to a function and let the function return the addition answer.
2. Given a string

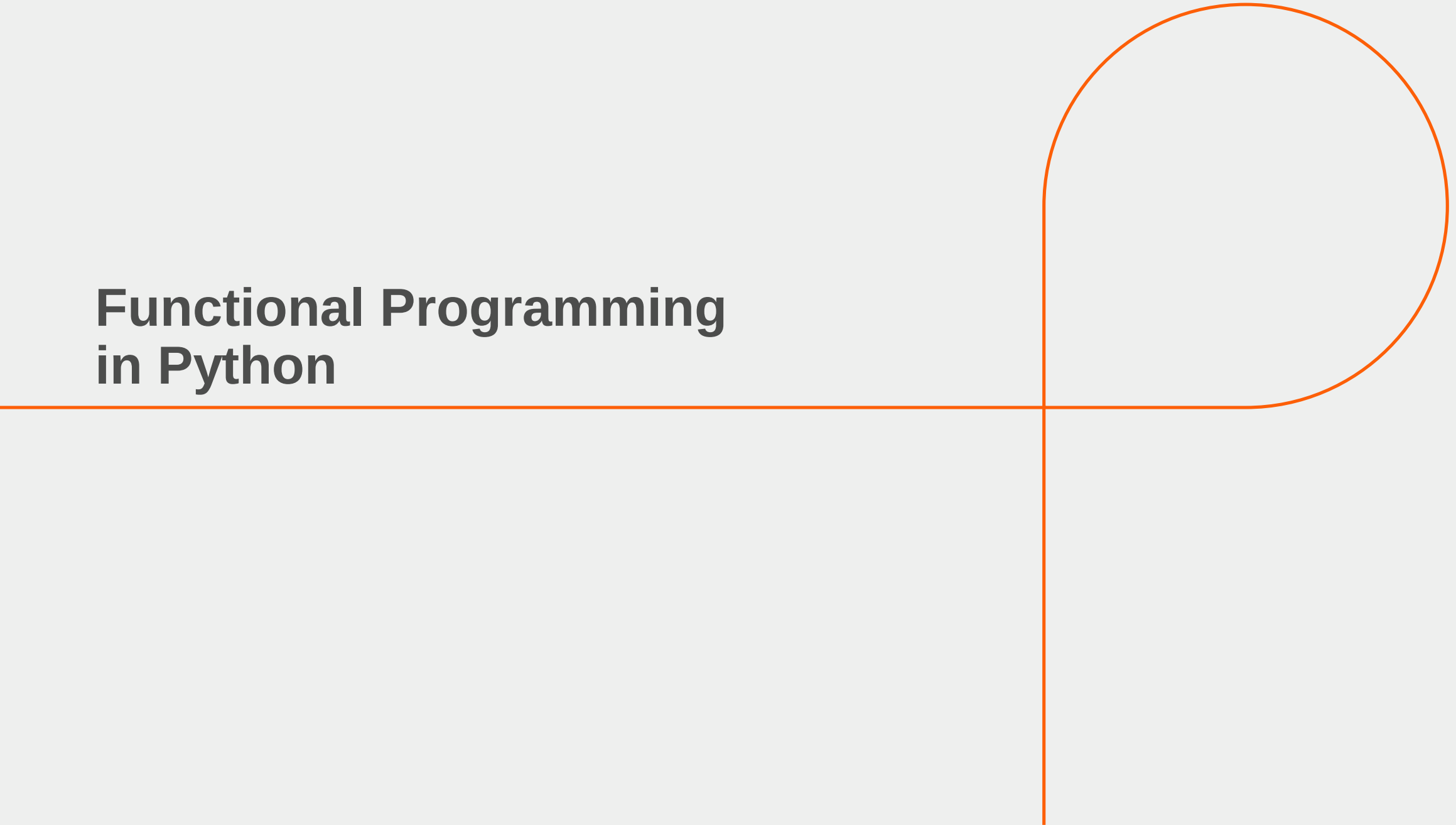
sentence = 'It is raining cats and dogs'

get 1 target list with length of each word in this sentence

Hint : Use map, lambda, split appropriately



Functional Programming in Python



Higher-order functions

map(func,seq) – for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.

```
def double(x): return 2*x

>>> lst = range(10)

>>> lst

[0,1,2,3,4,5,6,7,8,9]

>>> map(double,lst)

[0,2,4,6,8,10,12,14,16,18]
```

Higher-order functions: Filter

filter(boolfunc,seq) – returns a sequence containing all those items in seq for which boolfunc is True.

```
def even(x): return ((x%2 == 0)
```

highorder.py

```
>>> from highorder import *
>>> lst = range(10)
>>> lst
[0,1,2,3,4,5,6,7,8,9]
```

>>> filter(even,lst)

```
[0,2,4,6,8]
```


Persistent

Python’s higher-order functions

- Python supports higher-order functions that operate on lists similar to Scheme’s

```
>>> def square(x): return x*x
>>> def even(x): return 0 == x % 2
>>> map(square, range(10,20))
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
>>> filter(even, range(10,20))
[10, 12, 14, 16, 18]
>>> map(square, filter(even, range(10,20)))
[100, 144, 196, 256, 324]
```

- But many Python programmers prefer to use list comprehensions instead

List comprehensions

- A **list comprehension** is a programming language construct for creating a list based on existing lists
  – Haskell, Erlang, Scala and Python have them
- Why “comprehension”? The term is borrowed from math’s **set comprehension** notation for defining sets in terms of other sets
- A powerful and popular feature in Python
  – Generate a new list by applying a function to every member of an original list
- Python’s notation: **[expression** for **name** in **list]**

Persistent

List comprehensions

- The syntax of a list comprehension is somewhat tricky

  [x-10 **for** x **in** grades **if** x>0]

- Syntax suggests that of a **for**-loop, an **in** operation, or an **if** statement
- All three of these keywords (‘**for’, ‘in’,** and **‘if’**) are also used in the syntax of forms of list comprehensions

  [<u>expression</u> for **<u>name</u>** in **<u>list</u>]**

Persistent

List comprehensions

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

[**expression** for **name** **in** **list**]

Note: Non-standard colors on next few slides clarify the list comprehension syntax.

- Where **expression** is some calculation or operation acting upon the variable **name**
- For each member of the **list**, the list comprehension
  1. Sets **name** equal to that member
  2. Calculates a new value using **expression**
- It then collects these new values into a list which is the return value of the list comprehension

List comprehensions

- If **list** contains elements of different types, then **expression** must operate correctly on the types of all of **list** members
- If the elements of **list** are other containers, then the **name** can consist of a container of names that match the type and “shape” of the **list** members

```
>>> li = [(‘a’, 1), (‘b’, 2), (‘c’, 7)]

>>> [ n * 3 for (x, n) in li]

[3, 6, 21]
```

List comprehensions

- **expression** can also contain user-defined functions.

```
>>> def subtract(a, b): return a – b

>>> oplist = [(6, 3), (1, 7), (5, 5)]

>>> [subtract(y, x) for (x, y) in oplist]

[-3, 6, 0]
```


Persistent

Syntactic sugar

List comprehensions can be viewed as syntactic sugar for a typical higher-order functions

[**expression** for **name** in **list**]

map(lambda **name:** **expression**, **list**)

[**2*x+1** for **x** in **[10, 20, 30]**]

map(lambda **x:** **2*x+1**, **[10, 20, 30]**)

Persistent

Filtered list comprehension

- **Filter** determines whether **expression** is performed on each member of the **list.**
- For each element of **list**, checks if it satisfies the **filter condition.**
- If the **filter condition** returns **False**, that element is omitted from the list before the **list** comprehension is evaluated.

  [**expression** for **name** in **list** if **filter**]

Persistent

Filtered list comprehension (contd.)

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition
- So, only 12, 14, and 18 are produce

More syntactic sugar

Including an if clause begins to show the benefits of the sweetened form

[**expression** for **name** in **list** **if** **filt**]

map(lambda **name** **.** **expression**, filter(**filt, list**))

[**2*x+1** for **x** in **[10, 20, 30]** if **x > 0**]

map(lambda **x:** **2*x+1**,

filter(lambda **x: x > 0** **, [10, 20, 30]**)

Persistent

Nested list comprehensions

- Since list comprehensions take a list as input and produce a list as output, they are easily nested

```
>>> li = [3, 2, 4, 1]

>>> [elem*2 for elem in
        [item+1 for item in li] ]

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

- The inner comprehension produces: [4, 3, 5, 2]
- So, the outer one produces: [8, 6, 10, 4]

Syntactic sugar

[**e1** for **n1** in [**e1** for **n1** **list**]]

map(lambda **n1**: **e1**,

____map(lambda **n2: e2, list**))

[**2*x+1** for **x** in [**y*y** for **y** in **[10, 20, 30]**]]

map(lambda **x: 2*x+1,**

____map(lambda **y**: **y*y, [10, 20, 30]**))

Persistent

Small functions and the Lambda expression

If the function you need doesn’t exist, you need to write it. One way to write small functions is to use the lambda statement. lambda takes a number of parameters and an expression combining these parameters, and creates a small function that returns the value of the expression:

Examples:

- lowercase = lambda x: x.lower()
- print_assign = lambda name, value: name + '=' + str(value)
- adder = lambda x, y: x+y

Map and Lambda

Example:

```
Celsius = [39.2, 36.5, 37.3, 37.8]

print "Original temp in Celsius =", Celsius

Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)

print "Fahrenheit temp using map and lambda =", Fahrenheit

C = map(lambda x: (float(5)/9)*(x-32), Fahrenheit)

print "Celsius temp using map and lambda =", C
```

Filter and Lambda

Example:

```
r = [0, 1, 2, 3, 4, 5, 6]

target_list = filter(lambda x:x>3, r)

print "target list = "target_list

#target list = [4, 5, 6]


words =["abc","PSL", "XYZ","aaa","bbb"]

upperwords = filter(lambda str:str.isupper(),words)

print "Upperwords = ",upperwords

#Upperwords = ['PSL', 'XYZ']
```


Persistent

Assignments

1. Let’s say I give you a list saved in a variable: a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].

Write one line of Python that takes this list a and makes a new list b that has only the even elements of this list in it. (Use filter and lambda)

2. Given a string

sentence = 'It is raining cats and dogs'

get 1 target list with length of each word in this sentence

Hint: Use map, lambda, split appropriately

Summary :

With this we have come to an end of this session, where we discussed about ….

- Python Functions

In the next session we will discuss about

- Python Modules/Packages

Reference material

- http://www.tutorialspoint.com/python
- http://www.learnpython.org/

Persistent

Questions

Key contacts

Sakshi Jamgaonkar

sakshi_jamgaonkar@persistent.com

Asif Immanad

asif_immanad@persistent.co.in

Persistent

Persistent

Thank you!

Persistent University