Functional programming in Python

Yulia Newton, Ph.D.

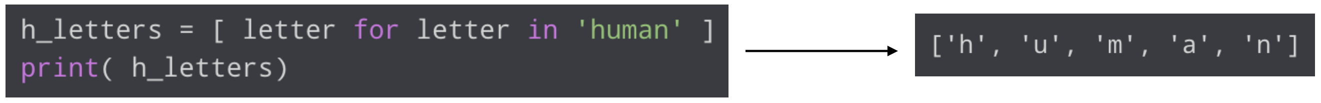
CS152, Introduction to Artificial Intelligence

San Jose State University

Spring 2022

List comprehension in Python

- List comprehension is about manipulating lists into other lists
- List comprehension is an example of declarative programming paradigm in Python
  - Functional programming
- We will cover Python list comprehension when we talk about declarative programming paradigms
  - Right now we are looking at Python from imperative programming point of view



```
h_letters = [ letter for letter in 'human' ]
print( h_letters)
```

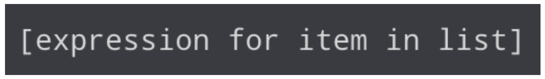
```
['h', 'u', 'm', 'a', 'n']
```

programiz.com

List comprehension in Python (cont’d)

List is the original list we want to transform into the new list

List comprehension syntax:



```
[expression for item in list]
```

programiz.com

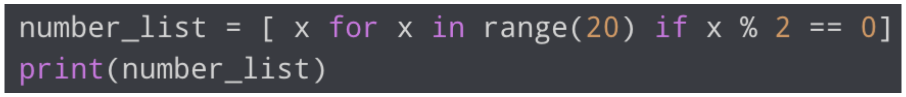
Item is each individual item in the original list

Expression is the expression we want to apply to the item; each item in the new list will be transformed by this expression to become an item in the new list

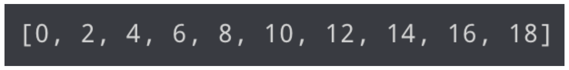
For is the iterator for the items in the original list

List comprehension in Python (cont’d)

Conditional statements in list comprehension:



```python
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list)
```

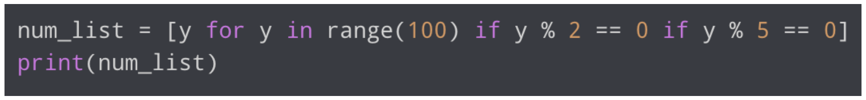


```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

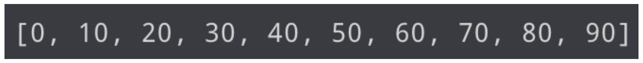
programiz.com

List comprehension in Python (cont’d)

Multiple conditional statements in list comprehension:

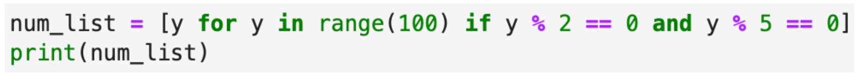


```python
num_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
print(num_list)
```



```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Alternatively could use *and* keyword:



```python
num_list = [y for y in range(100) if y % 2 == 0 and y % 5 == 0]
print(num_list)
```

programiz.com

List comprehension in Python (cont'd)

If-else in list comprehension:

```
obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
print(obj)
```

```
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']
```

programiz.com

List comprehension in Python (cont'd)

Using list comprehension to transpose matrices:

Let's say, we have this 2x4 matrix:

```python
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
```

1	2	3	4
4	5	6	8

We need to transpose this matrix into:

1	4
2	5
3	6
4	8

We could do the transpose with nested loops:

```python
transposed = []
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]

for i in range(len(matrix[0])):
    transposed_row = []

    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)

print(transposed)
```

```
[[1, 4], [2, 5], [3, 6], [4, 8]]
```

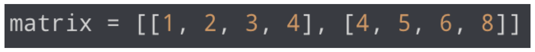
There is a more elegant solution that uses list comprehension:

```python
matrix = [[1, 2], [3,4], [5,6], [7,8]]
transpose = [[row[i] for row in matrix] for i in range(2)]
print (transpose)
```

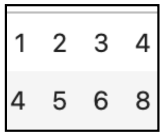
programiz.com

List comprehension in Python (cont'd)

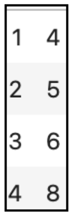
Using list comprehension to transpose matrices:

Let's say, we have this 2x4 matrix: 

```
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
```



1	2	3	4
4	5	6	8

We need to transpose this matrix into: 

1	4
2	5
3	6
4	8

programiz.com

List comprehension in Python (cont’d)

- List comprehension is an elegant way to define and create lists based on existing lists
  - User-friendly code and easier to read and understand than loops
- List comprehension code tends to be more compact and more efficient than normal functions and loops for creating list
- List comprehension does not produce side effects
- Every list comprehension code can be re-written with loops, however not every loop can be re-written with list comprehension

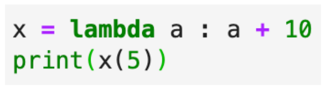
Lambda functions in python

- Remember, lambda functions are anonymous functions

Lambda function syntax in Python:

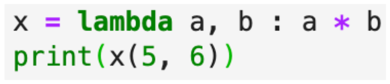
```
lambda arguments : expression
```

Lambda functions in python (cont’d)



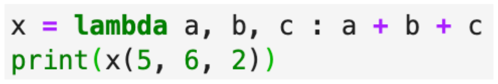
```python
x = lambda a : a + 10
print(x(5))
```

$\longrightarrow$ 15



```python
x = lambda a, b : a * b
print(x(5, 6))
```

$\longrightarrow$ 30

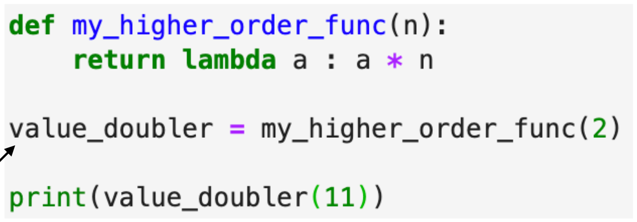


```python
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

$\longrightarrow$ 13

Lambda functions in python (cont’d)

Lambda functions can be first-class functions (returned from another function):



```python
def my_higher_order_func(n):
    return lambda a : a * n

value_doubler = my_higher_order_func(2)

print(value_doubler(11))
```

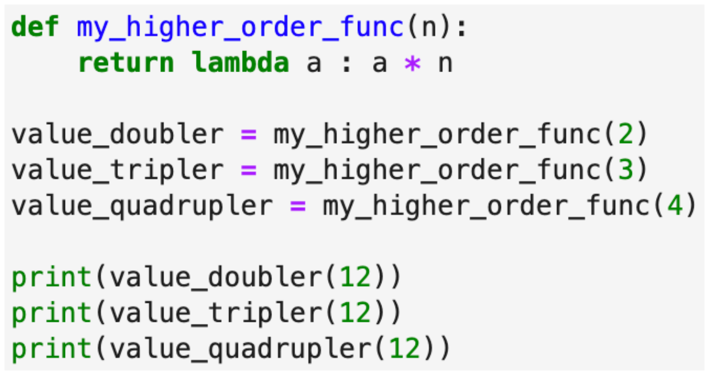
$\longrightarrow$

```
22
```

Notice that value_doubler is a closure

Lambda functions in python (cont'd)

my_higher_order_func() can be reused with different closures, providing versatile use for our lambda function:



```python
def my_higher_order_func(n):
    return lambda a : a * n

value_doubler = my_higher_order_func(2)
value_tripler = my_higher_order_func(3)
value_quadrupler = my_higher_order_func(4)

print(value_doubler(12))
print(value_tripler(12))
print(value_quadrupler(12))
```

$\longrightarrow$

```
24
36
48
```

Lambda functions in python (cont'd)

Can define and execute lambda function in the same expression:

```
(lambda x: x + 10)(5)
```

$\longrightarrow$

```
15
```

Lambda functions in python (cont'd)

We can get creative with our lambda functions:

```
person_name = lambda firstN, lastN: f"Full person's name: {firstN.upper()} {lastN.upper()}"
person_name('John', 'Smith')
```

$\longrightarrow$

```
"Full person's name: JOHN SMITH"
```

Placing ***f*** in front of a string in Python3 makes it a formatted string literal, also called f-string. In f-strings curly brackets contain expressions that can be evaluated and the values are placed into the appropriate positions in the string. The expressions are evaluated at runtime and then formatted using __format__ protocol. They are convenient programming shortcuts.

Lambda functions in python (cont'd)

Another more complex example:

```
# Functional way of finding sum of a list
import functools


mylist = [11, 22, 33, 44]

# Recursive Functional approach
def sum_the_list(mylist):

    if len(mylist) == 1:
        return mylist[0]
    else:
        return mylist[0] + sum_the_list(mylist[1:])

# lambda function is used
print(functools.reduce(lambda x, y: x + y, mylist))
```

$\longrightarrow$

```
110
```

geeksforgeeks.org

Map function in Python

- map() function returns a map object of the results after applying some function to each item of an iterable collection (list, tuple etc.)
  - Map object is an iterator
- Syntax:
  - map(function, iterable)

Map function in Python (cont'd)

Example of using map() in Python:

```python
# Python program to demonstrate working
# of map.

# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

$\longrightarrow$

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

geeksforgeeks.org

Map function in Python (cont’d)

We can use lambda functions with map():

```
# Double all numbers using map and lambda

numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))
```

$\longrightarrow$

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

geeksforgeeks.org

Map function in Python (cont’d)

A more complicated example that uses a lambda function with map():

```
# Add two lists using map and lambda

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

$\longrightarrow$

```
[5, 7, 9]
```

geeksforgeeks.org

Map function in Python (cont’d)

A more complicated example that uses a lambda function with map():

```
# List of strings
l = ['sat', 'bat', 'cat', 'mat']

# map() can listify the list of strings individually
test = list(map(list, l))
print(test)
```

$\longrightarrow$

```
[['s', 'a', 't'], ['b', 'a', 't'], ['c', 'a', 't'], ['m', 'a', 't']]
```

geeksforgeeks.org

Decorators in Python

- In Python a decorator is an implementation that allows adding behavior/functionality to a function or a class without changing the original implementation of that function/class
  - Remember the decorator design pattern in Java (typically covered in CS151 class)
- Decorators are wrappers for other functions and classes in order to add to their existing functionality without permanent changes to that functionality
- In Python, decorators are created using higher-order and first-class functions

Decorators in Python (cont'd)

Example of a decorator in Python:

```python
def lowercase(stringfunc):
    def wrapper():
        return stringfunc().lower()

    return wrapper

def say_hello():
    return 'HELLO WORLD'

decorate_hello = lowercase(say_hello)
print(decorate_hello())
```

lowercase() function accepts another function in as input parameter

wrapper() is a first-class function that invokes the function passed into its outer function

lowercase() returns *wrapper()*

$\longrightarrow$

```
hello world
```

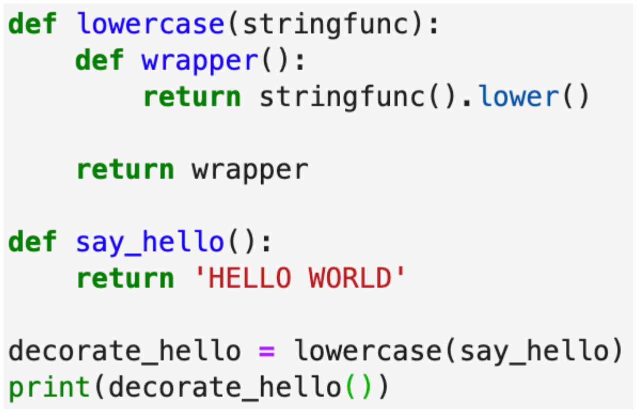
some other regular Python function

We decorate *say_hello()* by using higher-order function *lowercase()*

We execute our decorator

Decorators in Python (cont'd)

Example of a decorator in Python:



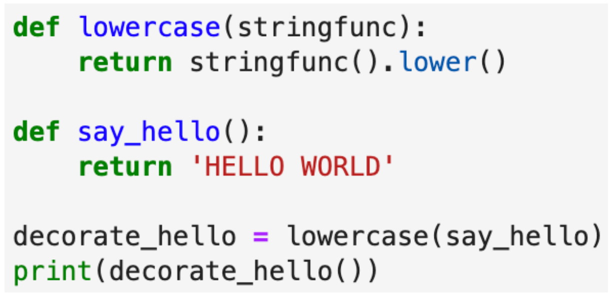
```python
def lowercase(stringfunc):
    def wrapper():
        return stringfunc().lower()

    return wrapper

def say_hello():
    return 'HELLO WORLD'

decorate_hello = lowercase(say_hello)
print(decorate_hello())
```

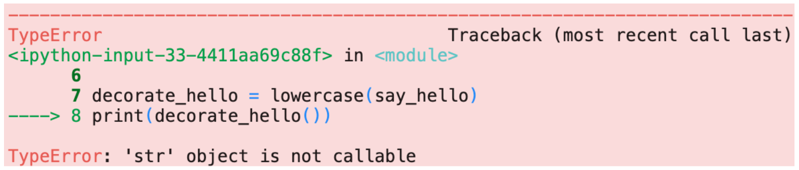
You might be wondering why do you need a *wrapper()* function at all:



```python
def lowercase(stringfunc):
    return stringfunc().lower()

def say_hello():
    return 'HELLO WORLD'

decorate_hello = lowercase(say_hello)
print(decorate_hello())
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-4411aa69c88f> in <module>
      6 
      7 decorate_hello = lowercase(say_hello)
----> 8 print(decorate_hello())

TypeError: 'str' object is not callable
```

Decorators in Python (cont'd)

Another decorator example:

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = my_decorator(say_whee)
```

$\longrightarrow$

```
>>> say_whee()
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

realpython.com

Decorators in Python (cont'd)

Another decorator example:

```python
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            pass  # Hush, the neighbors are asleep
    return wrapper

def say_whee():
    print("Whee!")

say_whee = not_during_the_night(say_whee)
```

If executed after 10pm, nothing happens (no output is produced):

$\longrightarrow$

```
>>> say_whee()
>>>
```

Decorators in Python (cont’d)

- Decorators wrap a function and modify its behavior
- Decorators return a new function with the new behavior without altering the behavior of the original function

Decorators in Python (cont’d)

- Python provides nice syntactic shortcuts to use decorators
  - Sometimes referred to as “pie syntax”
  - Using @ character is used to apply decorators to functions

Decorators in Python (cont'd)

Decorator example with using "pie syntax":

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
```

$\longrightarrow$

```python
say_whee()
```

```
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

Decorators in Python (cont’d)

Another decorator example with using “pie syntax”:

```python
def lowercase(stringfunc):
    def wrapper():
        return stringfunc().lower()

    return wrapper

@lowercase
def say_hello():
    return 'HELLO WORLD'

print(say_hello())
```

$\longrightarrow$

```
hello world
```

Decorators in Python (cont’d)

Another example:

```python
# importing libraries
import time
import math

# decorator to calculate duration
# taken by any function.
def calculate_time(func):

    # added arguments inside the inner1,
    # if function takes any arguments,
    # can be added like this.
    def inner1(*args, **kwargs):

        # storing time before function execution
        begin = time.time()

        func(*args, **kwargs)

        # storing time after function execution
        end = time.time()
        print("Total time taken in : ", func.__name__, end - begin)

    return inner1


# this can be added to any function present,
# in this case to calculate a factorial
@calculate_time
def factorial(num):

    # sleep 2 seconds because it takes very less time
    # so that you can see the actual difference
    time.sleep(2)
    print(math.factorial(num))

# calling the function.
factorial(10)
```

$\longrightarrow$

```
3628800
Total time taken in :  factorial 2.0061802864074707
```

geeksforgeeks.org

Decorators in Python (cont'd)

Reusing the decorator for different functions:

```python
def lowercase(stringfunc):
    def wrapper(*args, **kwargs):
        return stringfunc(*args, **kwargs).lower()

    return wrapper

@lowercase
def say_hello():
    return 'HELLO WORLD'

@lowercase
def personalized_hello(name):
    return f'HELLO {name}'

print(say_hello())
print(personalized_hello("ALEX"))
```

$\longrightarrow$

```
hello world
hello alex
```

Decorators in Python (cont'd)

Typical use of decorators in Python in real-life:

```python
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

realpython.com

Decorators in Python (cont'd)

Using decorators for debugging:

```python
import functools

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)           # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")           # 4
        return value
    return wrapper_debug
```

```python
@debug
def make_greeting(name, age=None):
    if age is None:
        return f"Howdy {name}!"
    else:
        return f"Whoa {name}! {age} already, you are growing up!"
```

```
>>> make_greeting("Benjamin")
Calling make_greeting('Benjamin')
'make_greeting' returned 'Howdy Benjamin!'
'Howdy Benjamin!'

>>> make_greeting("Richard", age=112)
Calling make_greeting('Richard', age=112)
'make_greeting' returned 'Whoa Richard! 112 already, you are growing up!'
'Whoa Richard! 112 already, you are growing up!'

>>> make_greeting(name="Dorrisile", age=116)
Calling make_greeting(name='Dorrisile', age=116)
'make_greeting' returned 'Whoa Dorrisile! 116 already, you are growing up!'
'Whoa Dorrisile! 116 already, you are growing up!'
```

realpython.com

In conclusion

- In this lecture module we saw how to use Python to perform functional programming
- Python is a versatile language that can be used to write code following many different programming paradigms