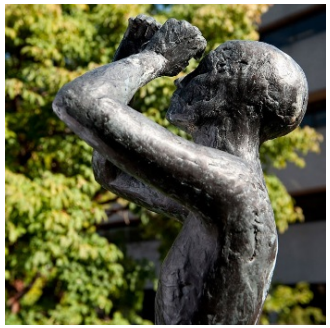


CS1117 – Introduction to Programming

Dr. Jason Quinlan,

School of Computer Science and Information Technology

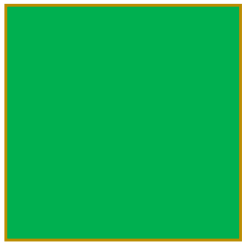
A TRADITION OF INDEPENDENT THINKING

UCC

University College Cork, Ireland

Coláiste na hOllscoile Corcaigh

ZIP



Zip – takes two equal length iterable collections

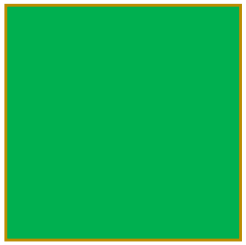
(list, string, tuple)

And merges them into a list of pairs of tuples:

List Example:

```
a = [1, 2, 3, 4, 5]
b = [2, 2, 9, 0, 9]
print(list(zip(a, b)))
# [(1, 2), (2, 2), (3, 9), (4, 0), (5, 9)]

print(zip(a, b))
# Python3 - <zip object at 0x105b49f50>
# Python 2 - [(1, 2), (2, 2), (3, 9), (4, 0), (5, 9)]
```

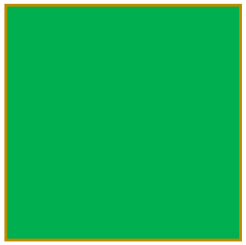
ZIP



String Example:

```
a = "hold"
b = "were"
print(list(zip(a, b)))
# [('h', 'w'), ('o', 'e'), ('l', 'r'), ('d', 'e')]

print(zip(a, b))
# Python3 - <zip object at 0x10a6f4e60>
# Python 2 - [('h', 'w'), ('o', 'e'), ('l', 'r'), ('d', 'e')]
```

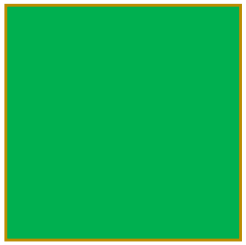
ZIP



Integer Example:

```
# integer example
a = 123
b = 456
print(list(zip(a, b)))
# zip argument #1 must support iteration

print(zip(a, b))
# Python3 - zip argument #1 must support iteration
# Python 2 - zip argument #1 must support iteration
```

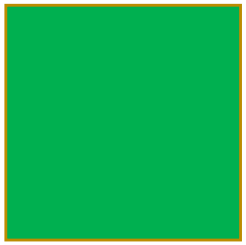
ZIP



Integer Example Fix:

```
# integer fix:
a = 123
b = 456
print(list(zip([a], [b])))
# [(123, 456)]

print(zip([a], [b]))
# Python3 - <zip object at 0x1094c8e10>
# Python 2 - [(123, 456)]
```

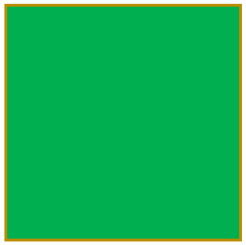
ZIP



Integer Example Fix:

```
# integer fix:
a = 123
b = 456
print(list(zip([a], [b])))
# [(123, 456)]

print(zip([a], [b]))
# Python3 - <zip object at 0x1094c8e10>
# Python 2 - [(123, 456)]
```

ZIP



Break String Example:

```
a = "hold"
b = "were"
print(list(zip([a], [b])))
# [('hold', 'were')]

print(zip([a], [b]))
# Python3 - <zip object at 0x10a6f4e60>
# Python 2 - [('hold', 'were')]
```

Question from Yesterday

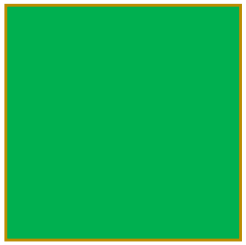
How do we get the max values from those integer?

When we pass them to zip?

```
a = 1537
b = 4264

print(list(zip([a], [b])))
# [(1537, 4264)]
```

Let’s see how…

MAP



Extracts each element in an iterable and passes it to a function

Parameters:

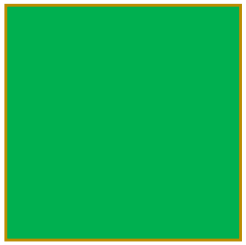
Function and an iterable collection

(really it can be viewed as a function and parameters)

Returns a list

```
# # # MAP # # #
a = [1, 2, 3, 4, 5]
b = [2, 2, 9, 0, 9]

print(map(max, a, b))
# [2, 2, 9, 4, 9]
```

MAP



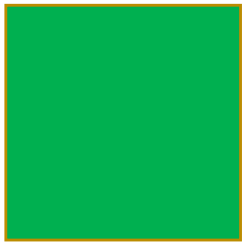
Take each of the values from ‘a’ and ‘b’

Pass them to the ‘max’ function

Store the returned ‘maximum value’ in a list

```
# # # MAP # # #
a = [1, 2, 3, 4, 5]
b = [2, 2, 9, 0, 9]

print(map(max, a, b))
# [2, 2, 9, 4, 9]
```

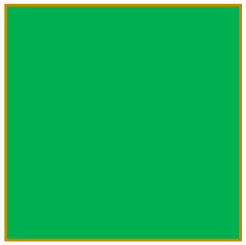
MAP



Similar to zip – Python3 assigns map results as an object

```
# # # MAP # # #
a = [1, 2, 3, 4, 5]
b = [2, 2, 9, 0, 9]

print(map(max, a, b))
# Python3 - <map object at 0x10a82c090>
# Python 2 - [2, 2, 9, 4, 9]

print(list(map(max, a, b)))
# Python3 - [2, 2, 9, 4, 9]
# Python 2 - [2, 2, 9, 4, 9]
```

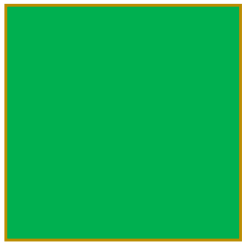
MAP



Map returns a result based on the smallest list only

```
a = [1, 2, 3]
b = [2, 2, 9, 0, 9]
print(list(map(max, a, b)))
# [2, 2, 9]

a = [1, 2, 3, 4, 5]
b = [2, 2, 9, 0]
print(list(map(max, a, b)))
# [2, 2, 9, 4]
```

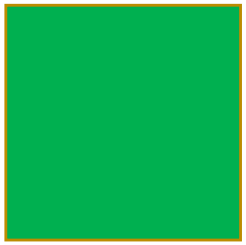
MAP



You can create your own functions and single parameters

```
def square(n):
    return(n*n)


a = [1, 2, 3, 4, 5]
print(list(map(square, a)))
# [1, 4, 9, 16, 25]
```

LAMBDA

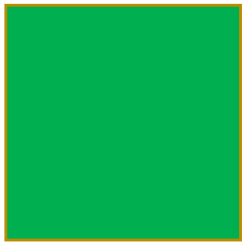


In Map we could call a function and pass each value in a list as individual parameters

```
def square(n):
    return(n*n)


a = [1, 2, 3, 4, 5]
print(list(map(square, a)))
# [1, 4, 9, 16, 25]
```

This function can be called from anywhere in our code

LAMBDA



In Python we can create a function that only exists for a single moment in time

For this we use the keyword ‘lambda’

‘lambda’ is a short way of creating an anonymous function (functions without a name)

Often used for a once-off function such as scenarios where you need to pass the result of a function as a parameter to another function.

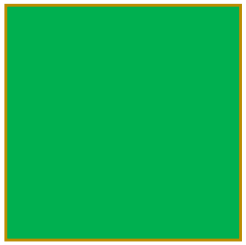
LAMBDA

If we were to rewrite the ‘square(n)’ function using ’lambda’

Using the format ‘lambda <parameter> : <expression>’

```
# # # # LAMBDA # # #
def square(n):
    return n*n



a = [1, 2, 3, 4, 5]
print(list(map(lambda n: n*n, a)))
# [1, 4, 9, 16, 25]
```

LAMBDA



Using the format ‘lambda <parameter> : <expression>’

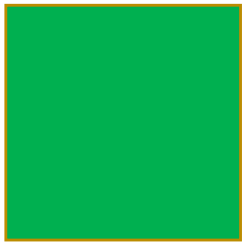
So remove the “def <name>” and “return”

Use ‘map()’ to extract each element from a and pass to lambda

Cast to ‘list()’ for Python3 compatibility

```
# # # # LAMBDA # # #
def square(n):
    return n*n


a = [1, 2, 3, 4, 5]
print(list(map(lambda n: n*n, a)))
# [1, 4, 9, 16, 25]
```

MAP

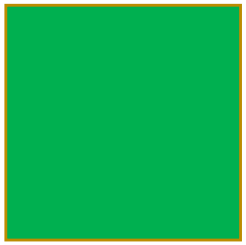


In Map we could call a function and pass each value in a list as individual parameters

But map expects a return value for every parameter

```
# # # # LAMBDA # # #
def square(n):
    return n*n


a = [1, 2, 3, 4, 5]
print(list(map(lambda n: n*n, a)))
# [1, 4, 9, 16, 25]
```

MAP

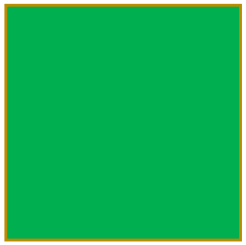


But what if we only want a subset of those parameters returned

Say, create a list of the odd numbers

```
def odd_numbers(a):
    alist = []
    for val in a:
        if val % 2 == 1:
            alist.append(val)
    return alist


a = [1, 2, 3]
print(odd_numbers(a))
# [1, 3]
```

MAP



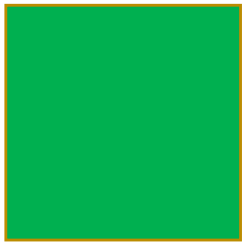
But what if we only want a subset of those parameters returned

Say, create a list of the odd numbers

```
def odd_numbers(val):
    # revise for MAP
    if val % 2 == 1:
        return val


a = [1, 2, 3]
print(list(map(odd_numbers, a)))
# [1, None, 3]
```

Map does not work…

FILTER



Extracts each element in an iterable and passes it to a function

Parameters:

Function and an iterable collection

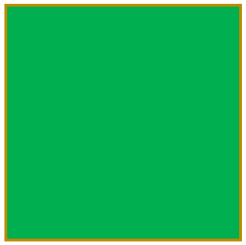
(really it can be viewed as a function and parameters)

FILTER – expects a Boolean return for each parameter passed

If the return is True the parameter is added to the list

If False, the parameter is dropped and not added to the list

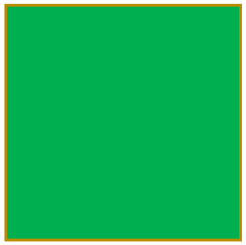
FILTER



So if we take our MAP version of “odd_numbers”

```
def odd_numbers(val):
    # revise for MAP
    if val % 2 == 1:
        return val


a = [1, 2, 3]
print(list(map(odd_numbers, a)))
# [1, None, 3]
```

FILTER

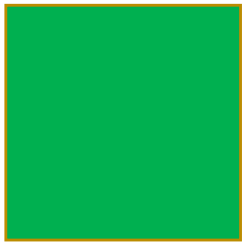


So if we take our MAP version of “odd_numbers”

Rewrite returning Booleans and call “filter(function, parameter_list)”

```
def odd_numbers(val):
    # revise for filter
    if val % 2 == 1:
        return True
    return False


# revise for filter
a = [1, 2, 3]
print(list(filter(odd_numbers, a)))
# [1, 3]
```

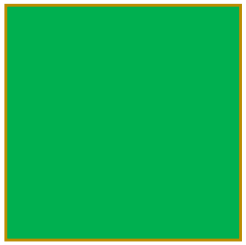
FILTER



Can you rewrite this using “lambda”?

```
def odd_numbers(val):
    # revise for filter
    if val % 2 == 1:
        return True
    return False


# revise for filter
a = [1, 2, 3]
print(list(filter(odd_numbers, a)))
# [1, 3]
```

FILTER

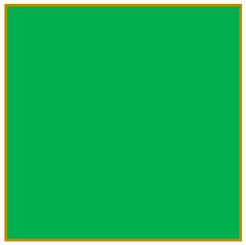


Can you rewrite this using “lambda”?

First let’s tidy up the Boolean returns…

```
def odd_numbers(val):
    # revise for filter
    return val % 2 == 1


# revise for filter
a = [1, 2, 3]
print(list(filter(odd_numbers, a)))
# [1, 3]
```

FILTER



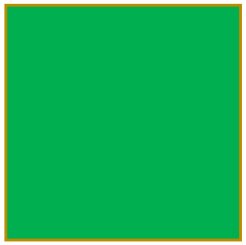
Can you rewrite this using “lambda”?

Then take out the function and add in lambda

```
def odd_numbers(val):
    # revise for filter
    return val % 2 == 1


# revise for filter
a = [1, 2, 3]
print(list(filter(odd_numbers, a)))
# [1, 3]
```

```
# revise for lambda
a = [1, 2, 3]
print(list(filter(lambda val: val % 2 == 1, a)))
# [1, 3]
```

REDUCE



Extracts each element in an iterable and passes it to a function

Parameters:

Function and an iterable collection

(really it can be viewed as a function and parameters)

REDUCE – expects a collection of parameter

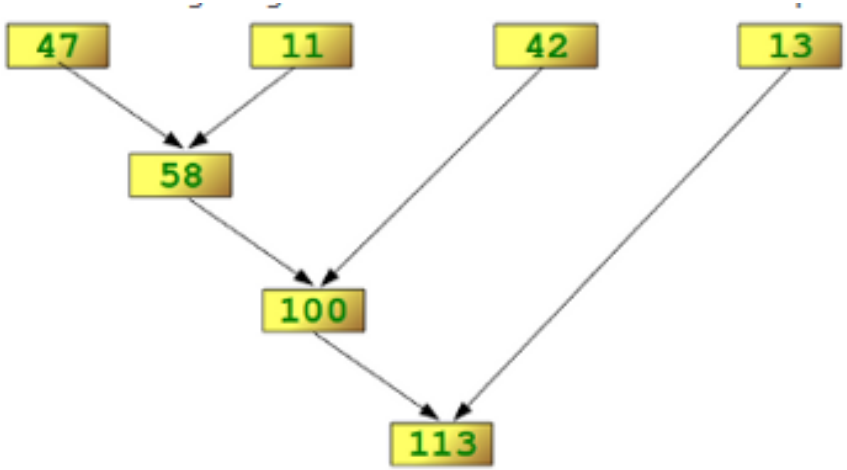
That it can reduce down to a single value

What is “reduce” but an accumulator ☺

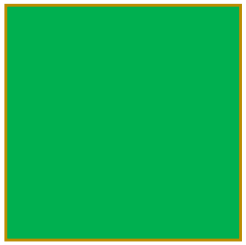
REDUCE

What is “reduce” but an accumulator ☺

```
>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
```



REDUCE

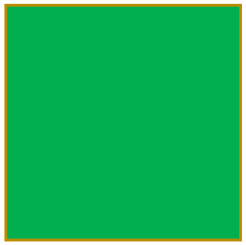


What is “reduce” but an accumulator ☺

Let’s look at this for Factorial from Lab2

```
def iter_factorial(n):

    accum = 1
    for val in range(2, n+1):
        # print(val)
        accum *= val

    return accum


print(iter_factorial(5))
# 120
```

REDUCE



What is “reduce” but an accumulator ☺

Let’s look at this for Factorial from Lab2

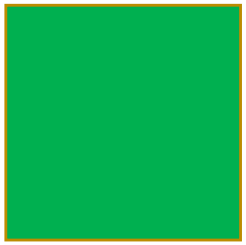
And using reduce?

```
print(functools.reduce(lambda x, y: x*y, range(1, 6)))
# python3 - 120
```

reduce() is a Python2 func, but the python3 version is functools.reduce()

So we need to “import functools”

REDUCE



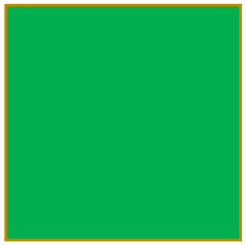
What is “reduce” but an accumulator ☺

Let’s look at this for Factorial from Lab2

And using reduce in a function?

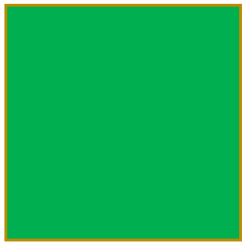
```
def iter_factorial(n):

    return functools.reduce(lambda x, y: x*y, range(1, n+1))



print(iter_factorial(5))
# 120
```

REDUCE



What is “reduce” but an accumulator ☺

Recap



We saw our starting function with its loops

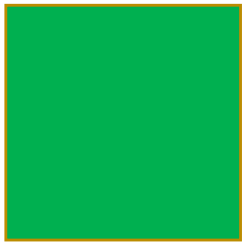
Which know we can typically cast the loops to a list comp

We can use zip if there are 2 list/string/tuple parameters and create a new list of pairs of values (tuples), based on index value

Map separates the one or more parameter containers into single values and send each individual value to a function

Map expects a return value for each passed parameter

Recap



lambda creates nameless functions (no def, <name> or return)

These function exist either as variables or as one time functions

filter separates a single parameter container into single values and send each value to a function, but only stores the sent value if a Boolean True is returned from the function

reduce also takes a single container, but acts as an accumulator, returning only a single value

Recap

Look at the past few Monday morning questions

Can we use what we learned this week for these questions?

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh