

**ECE 20875**

# Python for Data Science

**Chris Brinton and David Inouye**

**filters, map/reduce, and  
higher order functions**

# higher order functions

- Since functions are treated as first-class objects in Python, they can ...
- take one or more functions as arguments

```
def summation(nums):  
    return sum(nums)
```

```
def main(f, args):  
    result = f(args)  
    print(result)
```

```
if __name__ == "__main__":  
    main(summation, [1,2,3])
```

- return one or more functions

```
def add_two_nums(x, y):  
    return x + y
```

```
def add_three_nums(x, y, z):  
    return x + y + z
```

```
def get_appropriate(num_len):  
    if num_len == 3:  
        return add_three_nums  
    else:  
        return add_two_nums
```

- filter, map, and reduce are examples of built-in higher order functions

# list comprehension

- Simple way of creating a list based on an *iterable* Python object
- Elements in the new list are conditionally included and transformed as needed

```
[output expression for item in iterable if condition]
```
- An example:

```
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers if n > 2]
```
- Compared with a for loop
  - More computationally efficient
  - But less flexible!
- Can also have an if-else clause on the output expression

```
[output expression if-else clause for item in iterable condition(s) on iterable]
```
- Can use line breaks between brackets for readability

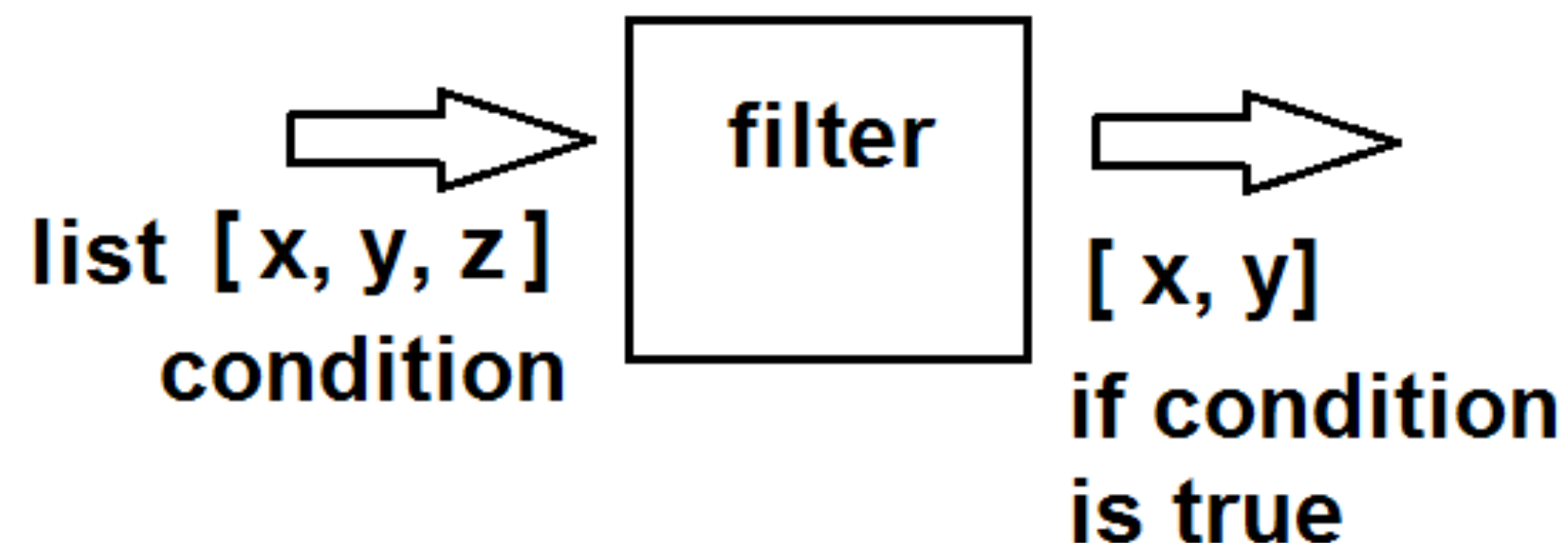
```
numbers = [1, 2, 3, 4, 5, 6, 18, 20]
squares = [
    "small" if number < 10 else "big"
    for number in numbers
    if number % 2 == 0
    if number % 3 == 0]
```
- Can also be nested

```
l = [['3', '4', '5'], ['6', '8', '10', '12']]
l2 = [[float(y) for y in x] for x in l]
```

# filter

- Remove undesired results from a list
- Needs two inputs:
  - (boolean) function to be carried out
  - Iterable (list) to be filtered

```
li = [5, 7, 22, 97, 54, 62, 77, 23,  
73, 61]  
final_list = list(filter(lambda x:  
(x%2 != 0) , li))  
print(final_list)
```



- The **lambda** function
  - Anonymous, i.e., without a name
  - Formatted as  
lambda arguments: expression
- Can have any number of arguments but only one expression

```
g = lambda x, y: x + y  
print(g(5,6))
```

# map

- Applies a function to all items in an input list (i.e., defines a mapping)
- Needs two inputs:
  - Function to apply
  - Iterable: A sequence, collection, or iterator object

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2,
items))
```

- Can also map e.g., a list of functions

```
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)
```

```
funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x:
x(i), funcs))
    print(value)
```

# reduce

- Perform computation on a list and return the (single value) result
- Rolling computation applied to sequential pairs of values
- Needs two inputs:
  - Function to apply
  - Sequence to iterate over

```
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y),
li)
```

- Can also define (non-anonymous) functions

```
def do_sum(x1, x2):
    return x1 + x2
reduce(do_sum, li)
```

- Operator functions can also be used

```
reduce(operator.add, li)
```

- Need to import the relevant modules

```
from functools import reduce
import operator
```