# Lists in Python

Thomas Schwarz SJ

# Lists

- Python is a high-level programming language with built-in sophisticated data structures
- The simplest of these data structures is the list.
- A list is just an <u>ordered</u> collection of other objects
    - The type of the objects is not restricted

- Let’s start unpacking this a bit.

# Lists

- We create a list by using the square brackets.
  - `alist = [1, 3.5, “hello”]`
    - A list with three elements of three different types
  - `blist = [1, 3.5, “hello”, 1]`
    - A list with four elements, where one element is repeated
  - `clist = [1, “hello”, 3.5]`
    - A different list than alist, but with the same elements
    - The order is different

# Lists

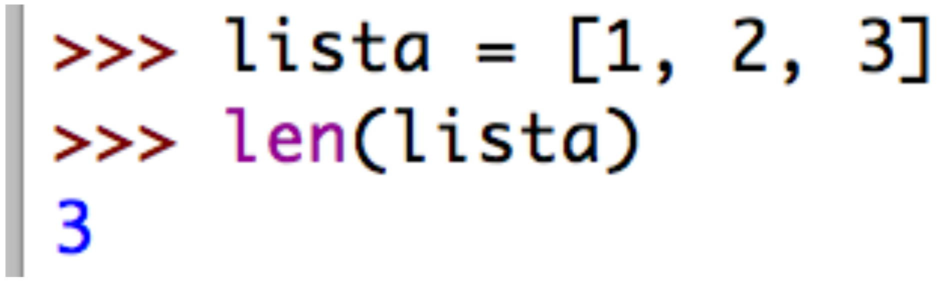
- Accessing elements in a list
    - We access elements in a list by using the square brackets and an index
    - Indices start at 0
- Example:
    - `lista = [‘a‘, ‘b’, ‘c‘, ‘d’]`
    - `lista[0]` is ‘a‘
    - `lista[1]` is ‘b’
    - `lista[2]` is ‘c’

# Lists

- Python uses negative numbers in order to count from the back of the list
  - `lista = [‘a‘, ‘b’, ‘c‘, ‘d’]`
  - `lista[-1]` is the last object, namely the character ‘d’
  - `lista[-2]` is the second-last object, namely the character ‘c’
  - `lista[-4]` is the first object, namely the character ‘a’

# Manipulating Lists

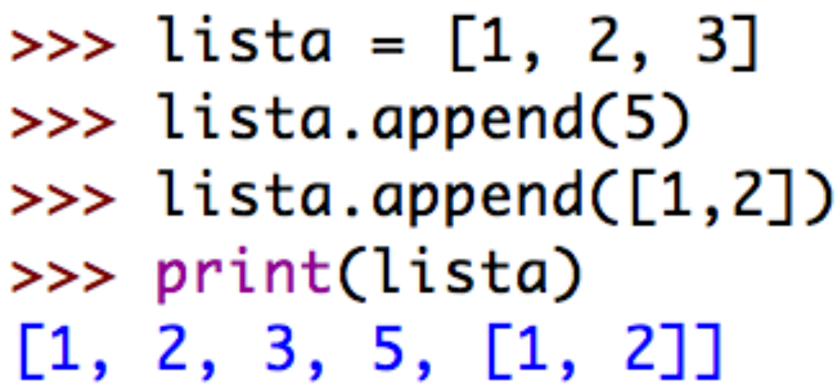
- We manipulate lists by calling list methods
  - You should read up on lists in the Python documentations
    - https://docs.python.org/3/tutorial/datastructures.html
- The length (number of objects in a list) is obtained by the len function.



```
>>> lista = [1, 2, 3]
>>> len(lista)
3
```

# Manipulating Lists

- We add to a list by using the append method
    - Example:

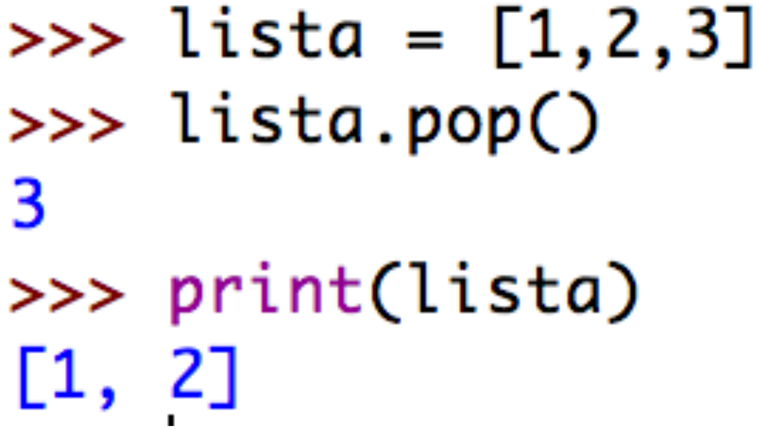


```
>>> lista = [1, 2, 3]
>>> lista.append(5)
>>> lista.append([1,2])
>>> print(lista)
[1, 2, 3, 5, [1, 2]]
```

- 
    - The resulting list `lista` has five elements, the last one being a list by itself.
- The append method always adds an element at the end.

# Manipulating Lists

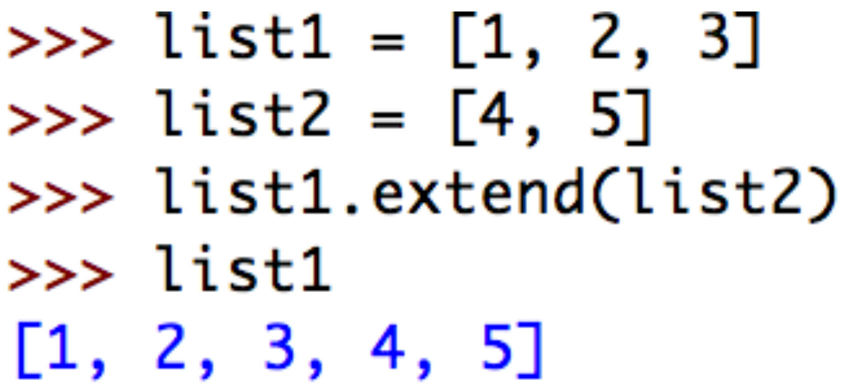
- The opposite of *append* is *pop*.
    - Whereas append returns the special object None, pop removes the last element in the list and returns it.
- Example



```
>>> lista = [1,2,3]
>>> lista.pop()
3
>>> print(lista)
[1, 2]
```

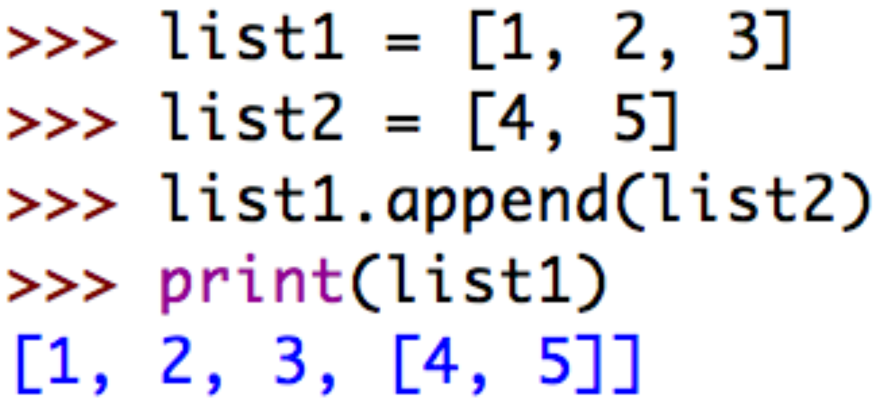
# Manipulating Lists

- We can also combine two lists with extend.
  - The method parameter is a list that is added to the first list.



```
>>> list1 = [1, 2, 3]
>>> list2 = [4, 5]
>>> list1.extend(list2)
>>> list1
[1, 2, 3, 4, 5]
```

  - This is different than appending.

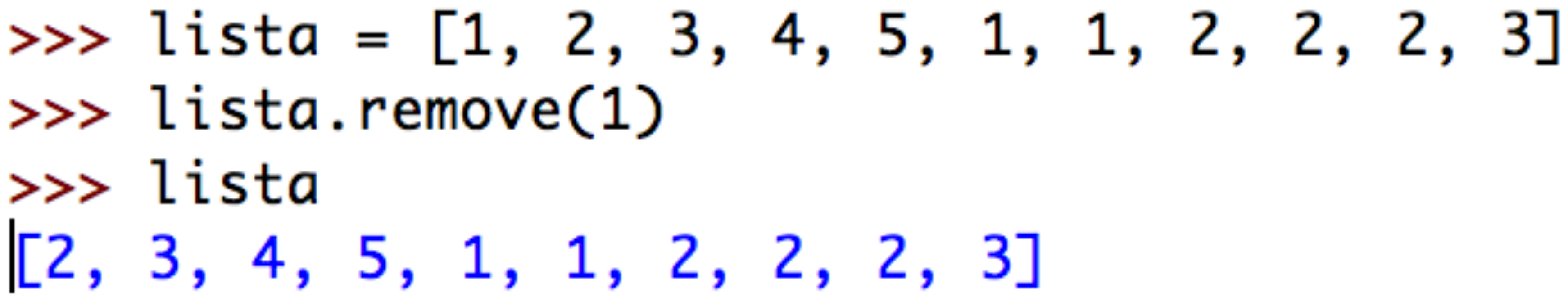


```
>>> list1 = [1, 2, 3]
>>> list2 = [4, 5]
>>> list1.append(list2)
>>> print(list1)
[1, 2, 3, [4, 5]]
```

  - The resulting list has four elements, with the last one being a list

# Manipulating Lists

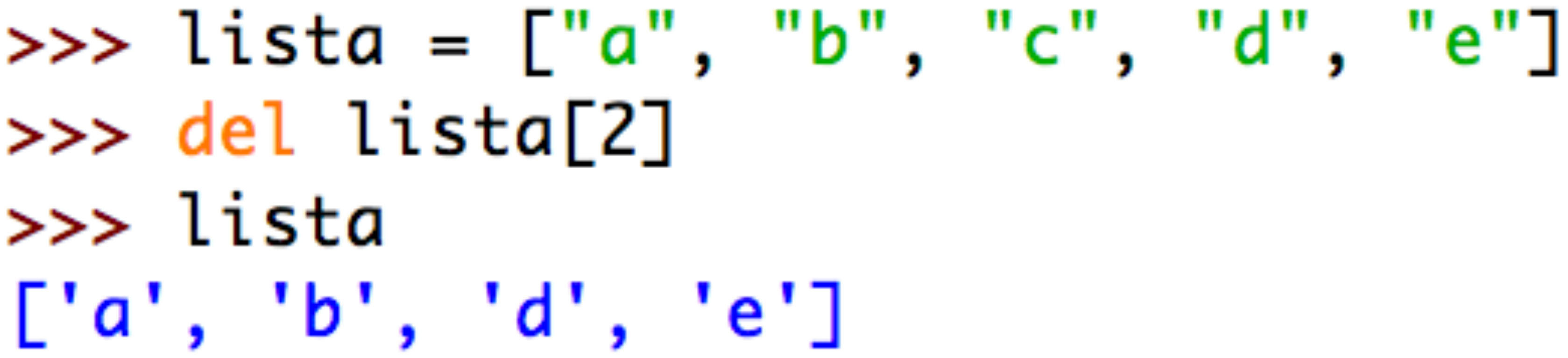
- To remove items from a list, we can use
  - remove
  - del
- The remove method removes the first element from the list that matches a parameter
  - It does not remove all elements
  - Example:



```
>>> lista = [1, 2, 3, 4, 5, 1, 1, 2, 2, 2, 3]
>>> lista.remove(1)
>>> lista
[2, 3, 4, 5, 1, 1, 2, 2, 2, 3]
```

# Manipulating Lists

- del operator:
  - A generic operator
  - In order to remove an item from a list, you specify a list and an index
    - Example: Remove the third element (“c”) from a list



```
>>> lista = ["a", "b", "c", "d", "e"]
>>> del lista[2]
>>> lista
['a', 'b', 'd', 'e']
```

# Manipulating Lists:
# A Standard Pattern

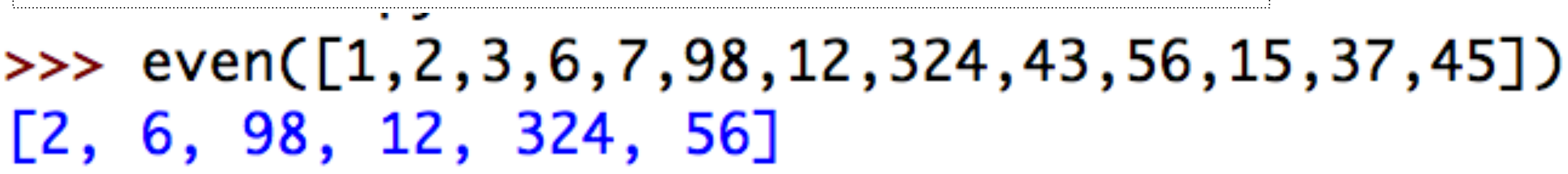
- A pattern for list modification
    - Often, we need to process a list
        - A standard pattern:
            - Create an empty result list
            - Walk through the processed list
            - Add elements to the result list

# Manipulating Lists:
# A Standard Pattern

- Example:
  - Filtering:
    - Retain all elements in a list that are even numbers

```
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

Create the result as an empty list



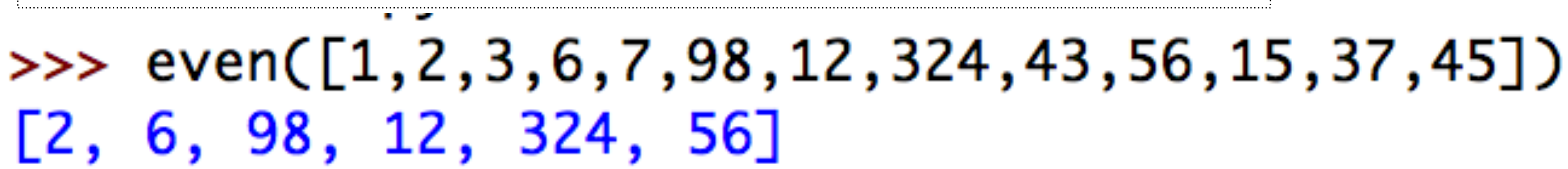
```
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```

# Manipulating Lists:
# A Standard Pattern

- Example:
  - Filtering:
    - Retain all elements in a list that are even numbers

```
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

Walk through the list



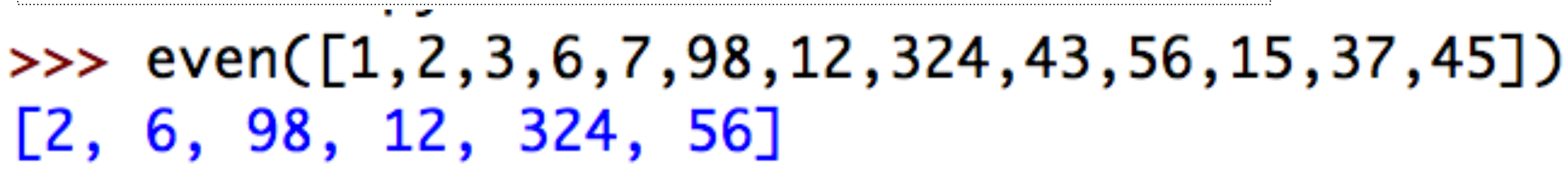
```
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```

# Manipulating Lists:
# A Standard Pattern

- Example:
  - Filtering:
    - Retain all elements in a list that are even numbers

```
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

Filter on condition



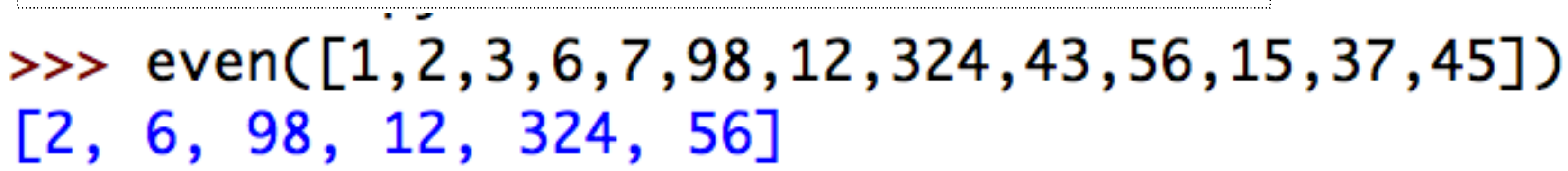
```
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```

# Manipulating Lists: A Standard Pattern

- Example:
  - Filtering:
    - Retain all elements in a list that are even numbers

```
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

Append to the result



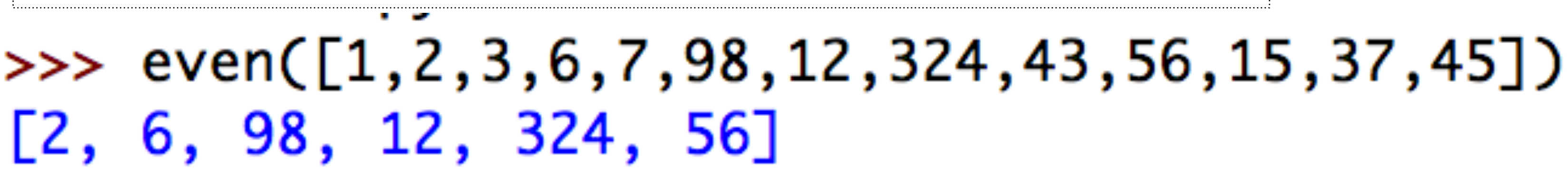
```
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```

# Manipulating Lists: A Standard Pattern

- Example:
  - Filtering:
    - Retain all elements in a list that are even numbers

```
def even(lista):
    result = []
    for ele in lista:
        if ele%2==0:
            result.append(ele)
    return result
```

Return the result



```
>>> even([1,2,3,6,7,98,12,324,43,56,15,37,45])
[2, 6, 98, 12, 324, 56]
```

# Manipulating Lists: A Standard Pattern

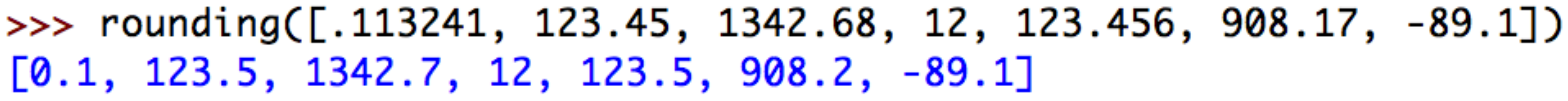
- Example:
  - Map — transforming all elements in a list
    - Given a list of numbers, round them to the nearest digit after the decimal point

# Manipulating Lists: A Standard Pattern

- Example:

```
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```

Create an empty list



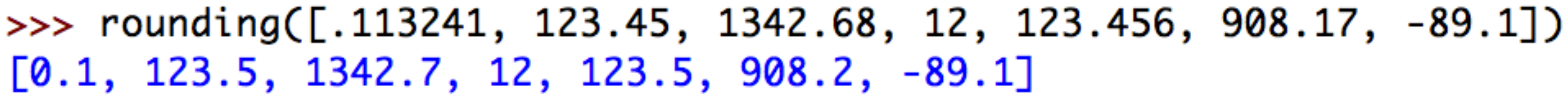
```
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```

# Manipulating Lists: A Standard Pattern

- Example:

```
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```

Walk through the list



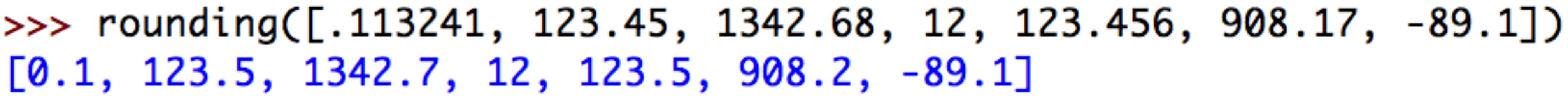
```
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```

# Manipulating Lists:
# A Standard Pattern

- Example:

```
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```

Apply the function to the list element



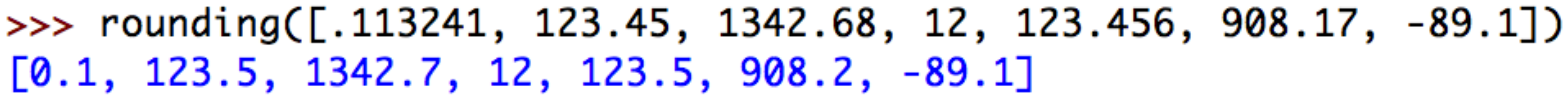
```
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```

# Manipulating Lists: A Standard Pattern

- Example:

Append to the result

```
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```



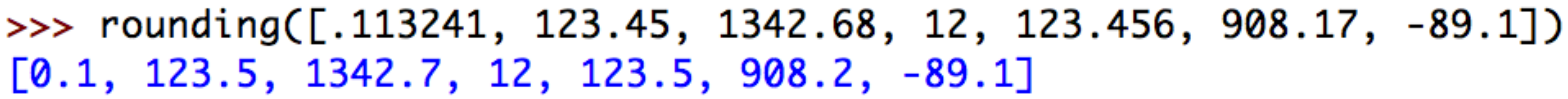
```
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```

# Manipulating Lists: A Standard Pattern

- Example:

Return the result

```
def rounding(lista):
    result = []
    for ele in lista:
        result.append(round(ele,1))
    return result
```



```
>>> rounding([.113241, 123.45, 1342.68, 12, 123.456, 908.17, -89.1])
[0.1, 123.5, 1342.7, 12, 123.5, 908.2, -89.1]
```

# Manipulating Lists: A Standard Pattern

- We can generate this example to all functions of list elements

```
def apply(function, lista):
    result = []
    for ele in lista:
        result.append(function(ele))
    return result
```

- This pattern is so important that Python 3 has a more elegant way of doing it. It is called list comprehension
    - The apply function was part of Python 2, depreciated in Python 2.3 and abolished in Python 3.5

# Lists are objects

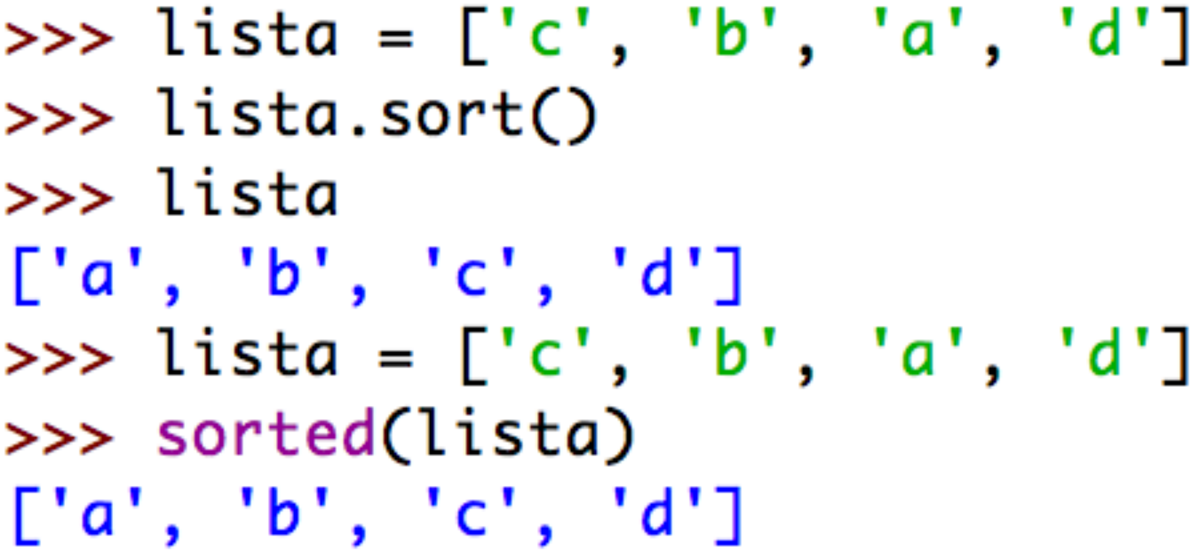
- Lists are objects
  - Objects have methods
    - Methods are functions that are called with an object as a parameter, but that are specific to the object
    - We write them as

      object . method ( additional, optional parameters )

    - In fact, method is a function and object is the first and sometimes only parameter

# Methods vs. Function

- There are two built-in ways to sort a list in Python:
  - The sorted function
  - The sort method for lists
- They are called differently because one is a method and one a function
  - sorted returns a sorted list
  - *.sort( ) does not return anything, but the list is sorted.



```
>>> lista = ['c', 'b', 'a', 'd']
>>> lista.sort()
>>> lista
['a', 'b', 'c', 'd']
>>> lista = ['c', 'b', 'a', 'd']
>>> sorted(lista)
['a', 'b', 'c', 'd']
```

# Manipulating Lists

- Here is an overview of the most important list methods:

| Method | Effect |
|---|---|
| append( ) | adds an element to the end of the list |
| clear( ) | removes all elements from a list |
| copy( ) | returns a copy of the list |
| count( ) | returns the number of elements in the list |
| extend( ) | adds the elements in the parameter to the list |
| index( ) | returns the index of the first occurrence of the parameter |
| insert( ) | inserts an element at the specified location |
| pop( ) | removes an element at the specified location or if left empty, removes the last element |
| remove( ) | removes the first element with that value |
| reverse( ) | reverses the order of the list |
| sort( ) | sorts the list |

# Range is not a list

- A range belongs to a data structure (called iterators) that are related to lists
  - In an iterator, you can always produce the next element
  - To make a list, just use the list keyword:

```
lista = list(range(2, 1000))
```

# Lists and for loops

- The for-loop in Python iterates through a list (or more generally an iterator)
    - `for x in lista:`
        - `x` takes on all values in `lista`

# Checking membership

- In Python, membership in a list is checked with the `in` keyword
  - There is a more appealing, alternative form of negation
- Examples:
  - `if element in lista:`
  - `if element not in lista:`
    - Use this one instead of the negation around the statement
      - `if not element in lista:`

# Sieve of Eratosthenes

- To calculate a list of all primes, we could:
    - Check all numbers in [2, 3, 4, … , *n*] that have no divisors
        - Which is tedious and does not scale to large *n*
    - Eliminate all multiples
        - This is the idea behind the famous Sieve of Eratostenes

# Sieve of Eratosthenes

- We start out with a list of all numbers between 2 and 1000
    - [2, 3, 4, 5, 6, 7, … , 999, 1000]
- The smallest number in the list is a prime, this would be 2
    - We can eliminate all true multiples of 2, that is, we remove 4, 6, 8, 10, … , 1000 from the list
    - This gives us
        - [2, 3, 5, 7, 9, 11, 13, …, 997, 999]
- The next smallest number has also to be a prime

# Sieve of Eratosthenes

  - [2, 3, 5, 7, 9, 11, 13, 15, 17, …, 997, 999]
- Therefore, 3, is a prime.
- For the next step, we eliminate all multiples of three that are left
  - [2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, … ,995, 997]
- We remove all multiples of 5 that remain in the list: 25, 35, 55, …
  - [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, … ,991, 997]
- And so we continue, until we can no longer eliminate multiples

# Sieve of Eratosthenes

- We implement this in Python
    - We first define a function that removes multiples of an element from a list (of numbers)
        - We need one parameter `limit` to tell us when we should stop

```
def remove_multiples(element, lista, limit):
    multiplier = 2
    while multiplier*element <= limit:
        if multiplier*element in lista:
            lista.remove(multiplier*element)
        multiplier += 1
```

# Sieve of Eratosthenes

- We can now implement the sieve
    - We initialize a list to the first 1000 elements
    - We maintain an index to tell us to which of the elements we already processed

```
def eratosthenes():
    lista = list(range(2, 1000))
    index = 0
```

# Sieve of Eratosthenes

- We stop when the index is about to fall out of the current size of the list
- Don’t forget to increase the index

```
def eratosthenes():
    lista = list(range(2, 1000))
    index = 0
    while index < len(lista):
        #Do the work here
       index += 1
```

# Sieve of Eratosthenes

- The work to do for each index is to remove the multiples of the current element

```
def eratosthenes(max_number):
    lista = list(range(2, max_number))
    index = 0
    while index < len(lista):
         element = lista[index]
         remove_multiples(element, lista, limit)
        index += 1
     return lista
```

# Sieve of Erathosthenes

- And here is the result, all primes until 1000

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61, 67, 71, 73,
77, 79, 83, 89, 91, 97, 101, 103, 107, 109, 113, 119, 121, 127, 131, 133, 137, 139,
143, 149, 151, 157, 161, 163, 167, 169, 173, 179, 181, 187, 191, 193, 197, 199, 203,
209, 211, 217, 221, 223, 227, 229, 233, 239, 241, 247, 251, 253, 257, 259, 263, 269,
271, 277, 281, 283, 287, 289, 293, 299, 301, 307, 311, 313, 317, 319, 323, 329, 331,
337, 341, 343, 347, 349, 353, 359, 361, 367, 371, 373, 377, 379, 383, 389, 391, 397,
401, 403, 407, 409, 413, 419, 421, 427, 431, 433, 437, 439, 443, 449, 451, 457, 461,
463, 467, 469, 473, 479, 481, 487, 491, 493, 497, 499, 503, 509, 511, 517, 521, 523,
527, 529, 533, 539, 541, 547, 551, 553, 557, 559, 563, 569, 571, 577, 581, 583, 587,
589, 593, 599, 601, 607, 611, 613, 617, 619, 623, 629, 631, 637, 641, 643, 647, 649,
653, 659, 661, 667, 671, 673, 677, 679, 683, 689, 691, 697, 701, 703, 707, 709, 713,
719, 721, 727, 731, 733, 737, 739, 743, 749, 751, 757, 761, 763, 767, 769, 773, 779,
781, 787, 791, 793, 797, 799, 803, 809, 811, 817, 821, 823, 827, 829, 833, 839, 841,
847, 851, 853, 857, 859, 863, 869, 871, 877, 881, 883, 887, 889, 893, 899, 901, 907,
911, 913, 917, 919, 923, 929, 931, 937, 941, 943, 947, 949, 953, 959, 961, 967, 971,
973, 977, 979, 983, 989, 991, 997]
```

# Sieve of Eratosthenes

- This implementation can be improved in a number of ways
    - For example, we do not need to remove all multiples because we know that some have been removed
        - For example, if we are processing 13, then we do no need to check for 2*13, 3*13, 4*13, … because they have already been replaced
- And there are ways to implement it more elegantly, but the point is just to see how to program with lists.

# $P \neq NP$

- Pythonic is not Non-Pythonic
  - Using indices when processing lists is usually not warranted
    - As much as possible, write functions on lists that would work with iterables just as well

# Python Iterators

- Python iterator: an object that contains a countable number of values
- An object is iterable if it implements an `iter` and a `next` method
  - `iter` returns an iterator
  - `next` gives us the next element.
    - When an iterator runs out of objects to provide on a next, it will create a StopIteration exception

# Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while num_iterator:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Creating an iterator

# Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while True:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Looping

# Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while True:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Getting the next item

# Python Iterators

```
numbers = [3,5,7,11,13,17,19,23,29,31]
num_iterator = iter(numbers)
while True:
    try:
        current_number = next(num_iterator)
        print(current_number)
    except StopIteration:
        break
```

Handling the exception generated when next fails

# Python Iterators

- Why do you need to know iterators:
  - To understand otherwise cryptic error messages
  - To use

# Python Generators

- Python allows you to define generators
  - We do not discuss generators in this course but you ought to be aware of their existence
- A generator object creates a sequence of objects
- A generator just creates a generator object
  - Looks like a function, but has a yield instead of a return

# Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

Generators look like functions !

# Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

But have a “yield” instead of a “return”

# Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

If this were a function, it would return just one element

# Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

But a generator keeps on yielding

# Python Generators

```
def fib_generator():
    previous, current = 0, 1
    while True:
        previous, current = current, previous+current
        yield current
```

**This is tuple assignment!**

Simultaneously assigns
previous <— current
current <— previous+current

# Python Generator

- This Python generator will generate all the Fibonacci numbers

# Tuples

Thomas Schwarz, SJ

# Tuples

- Tuples are like *immutable* lists.
    - They are immutable, i.e. you cannot change them once they have been created.
    - This allows us to use them as keys for a dictionary

# Tuple Creation

- You create a tuple by putting a comma separated list of items in parentheses

```
small_primes = (2,3,5,7,11,13)

digits = ("0", "1", "2", "3", "4", "5", "6", "7", "8", “9")
```

# Accessing Elements

- You access tuple coordinates by using the same notation as for lists

```
digits = ("0", "1", "2", "3", "4", "5", "6", "7", "8", "9"

print(digits[5])
```

  - prints out “5”

# Using Tuples: Tuple Assignment

- Tuple assignment
  - The “tuple operator” is the comma
    - Meaning, putting commas between things creates a tuple
    - Tuples can be assigned

# Using Tuples: Tuple Assignment

- Tuple assignment
  - The “tuple operator” is the comma
    - Meaning, putting commas between things creates a tuple
    - Tuples can be assigned as tuples
    - Which assigns the elements of the tuple as well
  - Example:

```
a, b = 3, 5
```

    - Creates two tuples and makes them equal
    - Result is a is 3 and b is 5

# Using Tuples: Tuple Assignment

- Tuple assignment makes it easy to switch values
    - Assume that we have two variables
    - We want them to exchange values
    - Here is code that does not succeed:

```
a=3
b=5

#now we want to switch values
a=b
b=a
print(a,b)  #prints 5 5
```

- Spend some time figuring out why

# Using Tuples: Tuple Assignment

- When we assign b=a, the old value of a has just be overwritten

```
a=3
b=5

#now we want to switch values
a=b
b=a
print(a,b)  #prints 5 5
```

# Using Tuples: Tuple Assignment

- We need to safeguard the value of *a* in a temporary variable
    - This is a well-known trap for beginners
    - But now we have three assignments

```
a=3
b=5

#now we want to switch values
temp = a
a=b
b=temp
print(a,b)  #prints 5 3
```

# Using Tuples: Tuple Assignment

- With tuples, this works much simpler

```
a=3
b=5

#now we want to switch values
a,b = b,a
print(a,b)  #prints 5 3
```

- The right side of the assignment is a tuple
- We assign it as a tuple to the left side
- Which then updates the values of a and b

# Using Tuples: Unpacking

- In general, you can *unpack* a tuple through an assignment
  - On the left, you have a tuple with variables
  - On the right, you have an established tuple

```
(name, last_name, birth_year, birth_month, birth_date) = caesar
```

  - This will load name, last_name, birth_year, … with the values in caesar
  - The number of elements on both sides of the assignment needs to be the same

# Using Tuples: Unpacking

- You can even unpack when calling a function
    - Put an asterisk before the tuple to cause the unpacking
        - Define a function of two variables

```
def geo_mean(a,b):
   return (a*b)**(1/2)
```

        - We call it in the usual way

```
print(geo_mean(4,7))
```

        - But we can also call it with a tuple

```
tp = (3,7)
print(geo_mean(*tp))
```

# Using Tuples: Several Return Values

- Assume that you want to return more than one value from a function
  - You can “kludge” it by return a list
    - Then you access the various return values via indices
  - You can return a tuple
    - And use tuple unpacking at the other end

# Using Tuples: Unpacking

- Several return values example
  - Assume that you want to return the mean and the standard deviation of a list of numbers

```
import math

def stats(lista):
    if not lista:       #lista is empty
        return 0,0
    mean = 0
    var = 0
    for element in lista:
        mean += element
    mean = mean/len(lista)
    for element in lista:
        var += (element-mean)**2
    return mean, math.sqrt(var/len(lista))
```

# Using Tuples: Unpacking

- This code returns a tuple

```
def stats(lista):
      …
      return mean/len(lista), math.sqrt(var/len(lista))
```

- If we call this function, we unpack in a single statement

```
mu, sigma = stats([12,23,12,12,14,12,13,16,29,11,12,13])
```