# CMPT 120
# Intro to CS & Programming I

## WEEK 9 (Mar. 10-14)

— *Jérémie O. Lumbroso* —

**Lecture 23:**
Common Programs on Lists

http://www.sfu.ca/~jlumbros/Courses/CMPT120/

# Thus Far, On Lists

We have seen

- what lists are
- they can be manipulated in a similar way to strings (index `[ ]` and slices `[ : ]`)
- we have seen how to write functions that take a list and return a value (`sum_of_list`, etc.)
- we have seen examples of functions that take a list and return a list (`J_list`)

# Checking a Property on a List

- Functions that go through an entire list to check whether a property is verified
  1. are the elements all integers?
  2. are the elements listed in increasing order?
  3. are the elements listed in "zig-zag" order?
  4. are the elements in Fibonacci's sequence (i.e., `L[i]==L[i-1]+L[i-2]`)?

- May be local properties (can check every element separately), or may require to keep track of the previous element, or several of the previous elements

# From English to Python

**Is a list increasing?**

- check if list is empty
  - if so then, **is increasing**
- save first element (which we can do because list is non-empty)
- for each x in the **rest** of the list
  - is x larger than the previous?
    - if not, **not increasing**
- when all elements have been check, **is increasing**

```
def is_increasing(L):
    if len(L) == 0:
        return True

    prev_elt = L[0]

    for x in L[1:]:
        if not (x > prev_elt):
            return False

    return True
```

Try with: `[1, 3, 4]` `[1, 4, 3]`

A Correct

B Problem

# Our Mistake

- We forgot to update the `prev_elt` variable
- It always contained the first element, and the comparison `not (x > prev_elt)` always checked if an element was larger, not than the previous, but than the first element

```
def is_increasing(L):
  if len(L) == 0:
    return True

  prev_elt = L[0]

  for x in L[1:]:
    if not (x > prev_elt):
      return False
    prev_elt = x


  return True
```

[1, 4, 3] returned True because:

- 4 is larger than 1
- 3 is larger than 1

must update `prev_elt`

# Correct Version

**Is a list increasing?**

- check if list is empty
  - if so then, **is increasing**
- save first element (which we can do because list is non-empty)
- for each x in the **rest** of the list
  - is x larger than the previous?
    - if not, **not increasing**
- when all elements have been check, **is increasing**

```
def is_increasing(L):
    if len(L) == 0:
        return True

    prev_elt = L[0]

    for x in L[1:]:
        if not (x > prev_elt):
            return False

    return True
```

Try with: `[1, 3, 4]` `[1, 4, 3]`

A Correct

B Problem

Python's Built-In List Operations, and how they modify lists

# OPERATIONS ON LISTS & THE MUTABILITY OF LISTS

# Python Built-In Functions on Lists

- Suppose `L` is a variable containing a list
- Built-in functions on a list are called just list the ones on strings: `L.thefunction(...)`
- Sometimes the function takes parameters
  - `L.count(elt)` which counts the number of times elt appears in the list L
- Sometimes the function takes **no** parameters
  - `L.reverse()` simply reverses the list

# help(list) — part I

```
|   append(...)
|       L.append(object) -- append object to end
|
|   count(...)
|       L.count(value) -> integer -- return number of occurrences
|       of value
|
|   extend(...)
|       L.extend(iterable) -- extend list by appending elements
|       from the iterable
|
|   index(...)
|       L.index(value, [start, [stop]]) -> integer -- return first
|       index of value.
|       Raises ValueError if the value is not present.
|
|   insert(...)
|       L.insert(index, object) -- insert object before index
|
```

# help(list) — part 2

```
|   pop(...)
|       L.pop([index]) -> item -- remove and return item at index
|       (default last).
|       Raises IndexError if list is empty or index is out of range.
|
|   remove(...)
|       L.remove(value) -- remove first occurrence of value.
|       Raises ValueError if the value is not present.
|
|   reverse(...)
|       L.reverse() -- reverse *IN PLACE*
|
|   sort(...)
|       L.sort(cmp=None, key=None, reverse=False) -- stable sort
|       *IN PLACE*;
|       cmp(x, y) -> -1, 0, 1
```

# Adding an Element to a List

```
>>> Lst1 = [ ]
>>> Lst2 = Lst1 + [3]
>>> Lst3 = Lst2.append(4)
```

We consider the expression above has been entered:

```
>>> print Lst2                    # what does Lst2 contain?
```

A [3] B [ [3 ] ] C Other D None E ERROR

```
>>> print Lst3                    # what does Lst3 contain?
```

A [3, 4] B [ 3, [4] ] C Other D None E ERROR

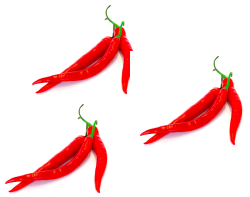
```
>>> Lst4 = Lst2
>>> Lst4.append(5)                # what does Lst4 contain?
```

A [3,4,5] B 2 C Other D 7 E ERROR

```
>>> print Lst2                    # what does Lst2 contain?
```

A [3] B [ [ 3 ] ] C Other D None E ERROR

# Mutability of Lists



- The built-in operations on the lists **modify the lists instead of returning a value**
- `L = []`
- This will not print anything, because the return value is None: `print L.append(4)`
- On the other hand L has been **modified** so `print L` will display `[4]`
- Important difference between modification and return value (like between **print** and **return**)

# Some Interesting Code

```
L1 = []
L2 = L1
L3 = L1[:]

L2.append(4)

print L1
print L2
print L3
```

Look at this on Python Tutor

# Non-Fruitful Functions

- **Non-Fruitful functions** are functions that do not return a value
- Remember what **return** means: if the function `f()` returns a value then when I do
  - `x = f()`
- `x` will contain that value
- If `f()` does not return a value,

# Fruitful or Non-Fruitful?

```
def fA():
  return 1
```

A Fruitful

B Non-fruitful

```
def fB():
  return
```

A Fruitful

B Non-fruitful

```
def fC():
  return "d".upper()
```

A Fruitful

B Non-fruitful

```
def fD():
  return
  return 4
```

A Fruitful

B Non-fruitful

```
def fE():
  print 4
```

A Fruitful

B Non-fruitful

```
def fF():
  print 4
  return 3
```

A Fruitful

B Non-fruitful

```
def fG():
  L = []
  return L + [4]
```

A Fruitful

B Non-fruitful

```
def fH():
  L = []
  return L.append(4)
```

A Fruitful

B Non-fruitful

# Pacing and Understanding

How well did you understand today?

A. Too easy, this lecture is way below my abilities

B. Everything went at a good pace, and I am fine

C. Too fast, but I will catch up on my own

D. Too fast, and I need you to slow down

E. I really do not think I can handle this

# Who Plans on Going to Tutorial?

**Tutorial**: today from 12:30pm to 2:30pm in TASC 1 9204

- **A** I am coming
- **B** I would like to come but I cannot
- **C** I do not find these tutorial sessions helpful
- **D** I do not need help, I am fine
- **E** Three hours a week is waaaaay more than enough time to be spending with you — no offense