Wikimedia Commons

BBM 101

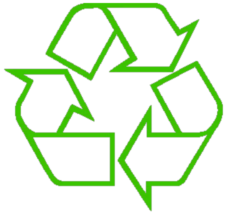
Introduction to Programming I

Lecture #06 – Arrays, Lists, Tuples

HACETTEPE UNIVERSITY

Fuat Akal, Aykut Erdem & Erkut Erdem // Fall 2019

Last time... **Control Flow, Functions**



Repeating yourself

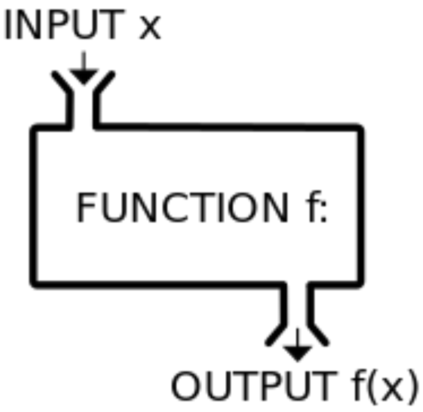
```
for f in [30,40,50]:
    print(f,(f-32)/9.0*5)
```

```
counter = 1
while counter <= n:
    s = s + counter
    counter += 1
```

Making decisions

```
if val < 0:
    result = - val
else:
    result = val
```

```
if height > 100:
   print("space")
elif height > 50:
   print("mesosphere")
elif height > 20:
   print("stratosphere")
else:
   print("troposphere")
```



Functions

```
def dbl_plus(x):
    return 2*x + 1
```

Lecture Overview

- Arrays
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

We will cover these later.

Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

Data Structures

- A *data structure* is way of organizing data
  - Each data structure makes certain operations convenient or efficient
  - Each data structure makes certain operations inconvenient or inefficient

- Example: What operations are efficient with:
  - a file cabinet sorted by date?
  - a shoe box?

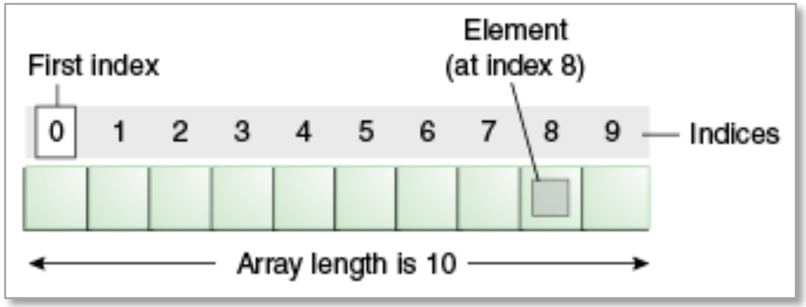
Lecture Overview

- Arrays
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

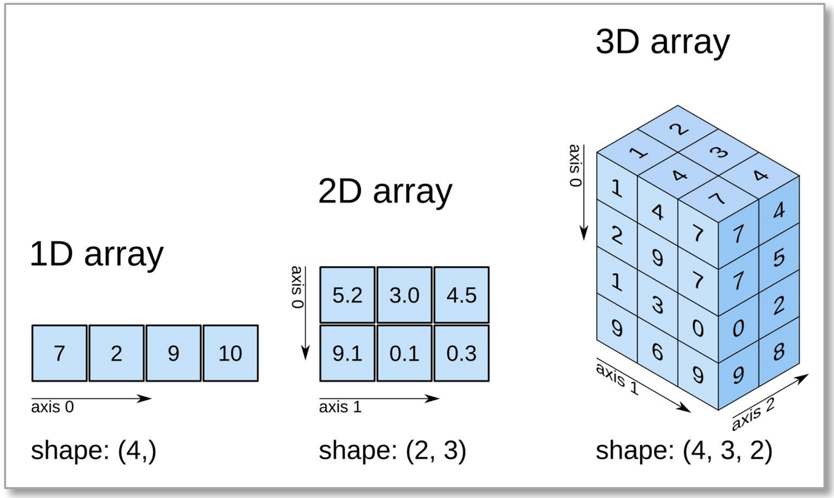
Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

An Array is …

- a container which can hold a **fix** number of items and these items should be of the **same** type.
  - Each item stored in an array is called an **element**.
  - Each location of an element in an array has a numerical **index**, which is used to identify the element.



Wait for ***Understanding Data*** lecture (Week 13) to learn more about arrays.



Lecture Overview

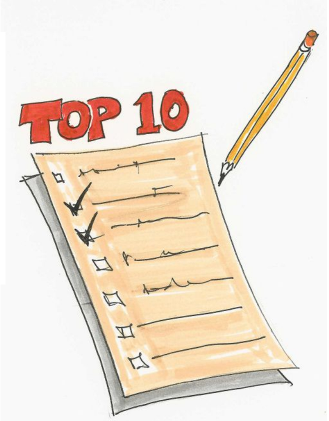
- Arrays
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

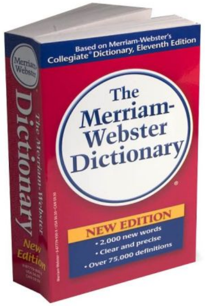
Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe’s CSE 140 class

A Collection Groups Similar Things

- List: ordered
- Set: unordered, no duplicates
- Tuple: unmodifiable list
- Dictionary: maps from values to values

  Example: word → definition




Based on Merriam-Webster's Collegiate Dictionary, Eleventh Edition
The Merriam-Webster Dictionary
NEW EDITION
• 2,000 new words
• Clear and precise
• Over 75,000 definitions
Dictionary

Lecture Overview

- Arrays
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

What is a List?

- A list is an ordered sequence of values, where each value is identified by an index.

- What operations should a list support efficiently and conveniently?
  - Creation
  - Querying/Lookup
  - Mutation

List Creation

- Use square brackets to specify a list.
- Separate each element with a comma.

```
a = [3, 4, 5]

b = [ 5, 3, 'hi' ]

c = [ 4, 'a', a ]

d = [ 3, 1, 2*2, 1, 10/2, 10-1 ]

e = []     # empty list
```

List Creation: Example - 1

```
L = ['I did it all', 4, 'love']

for i in range(len(L)):
   print(L[i])
```

```
>> I did it all
>> 4
>> love
```

List Creation: Example - 2

```
Techs = ['MIT', 'Caltech']
Ivys = ['Harvard', 'Yale', 'Brown']
Univs = [Techs, Ivys]
Univs1 = [['MIT','Caltech'],['Harvard','Yale','Brown']]

print('Univs =', Univs)
print('Univs1 =', Univs1)
print(Univs == Univs1)
```

```
>> Univs = [['MIT','Caltech'],['Harvard','Yale','Brown']]
>> Univs1 = [['MIT','Caltech'],['Harvard','Yale','Brown']]
>> True
```

How to Evaluate a List Expression

- `[a, b, c, d]` list creation
  - To evaluate:
    - evaluate each element to a value, from left to right
    - make a list of the values
  - The elements can be arbitrary values, including lists
    - `["a", 3, 3.14*r*r, fahr_to_cent(-40), [3+4, 5*6]]`

Same tokens "`[]`" with two *distinct* meanings

List expression

- `a[b]` list indexing or dereferencing
  - To evaluate:
    - evaluate the list expression to a value
    - evaluate the index expression to a value
    - if the list value is not a list, execution terminates with an error
    - if the element is not in range (not a valid index), execution terminates with an error
    - the value is the given element of the list value (counting from zero)

Index expression

List Expression Examples

What does this mean (or is it an error)?

```
["four", "score", "and", "seven", "years"][2]

["four", "score", "and", "seven", "years"][0,2,3]

["four", "score", "and", "seven", "years"][[0,2,3]]

["four", "score", "and", "seven", "years"][[0,2,3][1]]
```

List Expression Examples

```
>>> ["four", "score", "and", "seven", "years"][2]
'and’

>>> ["four", "score", "and", "seven", "years"][0,2,3]
TypeError: list indices must be integers or slices, not tuple

>>> ["four", "score", "and", "seven", "years"][[0,2,3]]
TypeError: list indices must be integers or slices, not list

>>> ["four", "score", "and", "seven", "years"][[0,2,3][1]]
'and'
```

List Lookup

- Extracting part of the list:
  - Single element: **`mylist[index]`**
  - Sublist (“slicing”): **`mylist[startidx : endidx]`**

- Find/lookup in a list
  - **`x in mylist`**
    - Evaluates to a boolean value

  - **`mylist.index(x)`**
    - Return the int index in the list of the first item whose value is x. It is an error if there is no such item.

  - **`list.count(x)`**
    - Return the number of times x appears in the list.

List Lookup: Exercise

```
def index(somelist, value):
   """Return the position of the first occurrence of
      the element value in the list somelist.
      Return None if value does not appear in
      somelist."""
  i = 0
  for c in somelist:
    if c == value:
      return i
    i = i + 1
  return None
```

```
gettysburg = ["four", "score", "and",
               ”seven", "years", "ago"]
index(gettysburg, "and")       # 2
index(gettysburg, "years”)     # 4
gettysburg.count('seven')      # 1
```

List Mutation

- Insertion
- Removal
- Replacement
- Rearrangement

List Insertion

- **`mylist.append(x)`**
  - Extend the list by inserting x at the end

- **`mylist.extend(L)`**
  - Extend the list by appending all the items in the argument list

- **`mylist.insert(i, x)`**
  - Insert an item before a given position.
  - `a.insert(0, x)` inserts at the front of the list
  - `a.insert(len(a), x)` is equivalent to `a.append(x)`

List Insertion: Examples

Python statement	Content of list1
`>>> list1 = [1, 2, 3]`	`[1, 2, 3]`
`>>> list1.append(4)`	`[1, 2, 3, 4]`
`>>> list1.insert(2, 5)`	`[1, 2, 5, 3, 4]`
`>>> list2 = [10, 20]`	
`>>> list1.extend(list2)`	`[1, 2, 5, 3, 4, 10, 20]`
`>>> list1.append(list2)`	`[1, 2, 5, 3, 4, 10, 20, [10, 20]]`

```
>>> list1[7]
[10, 20]
>>> list1[7][0]
10
>>> list1[7][1]
20
```

List Removal

- **`list.remove(x)`**
  - Remove the first item from the list whose value is x
  - It is an error if there is no such item

- **`list.pop([i])`**
  - Remove the item at the given position in the list, and return it.
  - If no index is specified, a.pop() removes and returns the last item in the list.

Notation from the Python Library Reference:
The square brackets around the parameter, “[i]”, means the argument is *optional.*
It does *not* mean you should type square brackets at that position.

List Removal - Examples

Python statement	Content of list1
`>>> list1 = [1, 2, 3]`	`[1, 2, 3]`
`>>> list1.remove(2)`	`[1, 3]`
`>>> list2 = list1.copy()`	
`>>> list1.extend(list2)`	`[1, 3, 1, 3]`
`>>> list1.remove(3)`	`[1, 1, 3]`
`>>> list1.pop()`	`[1, 1]`

How can you remove all occurences of an element?

List Replacement

- **`mylist[index] = newvalue`**

- **`mylist[start : end] = newsublist`**
  - Can change the length of the list
  - `start` is inclusive, `end` is not
  - `mylist[ start : end ] = [ ]` # removes multiple elements
  - `a[len(a):] = L` # is equivalent to a.extend(L)

List Replacement - Examples

Python statement	Content of list1
`>>> list1 = [1, 2, 3]`	`[1, 2, 3]`
`>>> list1[len(list1)-1] = 9`	`[1, 2, 9]`
`>>> list2 = list1`	
`>>> list1[1:2] = list2`	`[1, 1, 2, 9, 9]`
`>>> list1[1:3] = list2`	`[1, 1, 1, 2, 9, 9, 9, 9]`
`>>> list2[3:8] = []`	`[1, 1, 1]`
`>>> list2 = [5, 6]`	`[1, 1, 1]`

List Slicing

`mylist[startindex : endindex]` evaluates to a sublist of the original list

- **`mylist[index]`** evaluates to an element of the original list

- Arguments are like those to the **`range`** function
  - **`mylist[start : end : step]`**
  - start index is inclusive, end index is exclusive
  - *All* 3 indices are *optional*

- Can assign to a slice: **`mylist[s : e] = yourlist`**

List Slicing: Examples

```
test_list = ['e0', 'e1', 'e2', 'e3', 'e4', 'e5', 'e6']
```

From e2 to the end of the list:	`test_list[2:]`
From beginning up to (but not including) e5:	`test_list[:5]`
Last element:	`test_list[-1]`
Last four elements:	`test_list[-4:]`
Everything except last three elements:	`test_list[:-3]`
Reverse the list:	`test_list[::-1]`
Get a copy of the whole list:	`test_list[:]`

List Rearrangement

- **`list.sort()`**
  - Sort the items of the list, in place.
  - "in place" means by modifying the original list, not by creating a new list.

- **`list.reverse()`**
  - Reverse the elements of the list, in place.

Sorting

```
hamlet = "to be or not to be that is the
                                    question".split()
print("hamlet:", hamlet)

print("sorted(hamlet):", sorted(hamlet))

print("hamlet:", hamlet)

print("hamlet.sort():", hamlet.sort())
print("hamlet:", hamlet)

print("hamlet.reverse():", hamlet.reverse())
print("hamlet:", hamlet)
```

Sorting

```
hamlet: ['to', 'be', 'or', 'not', 'to', 'be', 'that', 'is',
'the', 'question']
```

```
sorted(hamlet): ['be', 'be', 'is', 'not', 'or', 'question',
'that', 'the', 'to', 'to']
```

```
hamlet: ['to', 'be', 'or', 'not', 'to', 'be', 'that', 'is',
'the', 'question']
```

```
hamlet.sort(): None
```

```
hamlet: ['be', 'be', 'is', 'not', 'or', 'question', 'that',
'the', 'to', 'to']
```

```
hamlet.reverse(): None
```

```
hamlet: ['to', 'to', 'the', 'that', 'question', 'or', 'not',
'is', 'be', 'be']
```

Customizing the Sort Order

Goal: sort a list of names ***by last name***

```
names = ["Isaac Newton", "Albert Einstein", "Niels
Bohr", "Marie Curie", "Charles Darwin", "Louis
Pasteur", "Galileo Galilei", "Margaret Mead"]
```

```
print("names:", names)
```

This does NOT work:

```
print("sorted(names):", sorted(names))
```

When sorting, how should we compare these names?

```
"Niels Bohr"
"Charles Darwin"
```

```
sorted(names): ['Albert Einstein', 'Charles
Darwin', 'Galileo Galilei', 'Isaac Newton',
'Louis Pasteur', 'Margaret Mead', 'Marie
Curie', 'Niels Bohr']
```

Sort Key

A sort key is a different value that you use to sort a list, instead of the actual values in the list

```
def last_name(str):
    return str.split(" ")[1]

print('last_name("Isaac Newton"):', 
last_name("Isaac Newton"))
```

Two ways to use a sort key:

1. Create a new list containing the sort key, and then sort it
2. Pass a key function to the sorted function

1. Use a sort key to create a new list

Create a different list that contains the sort key, sort it, then extract the relevant part:

```
names = ["Isaac Newton", "Fre
# keyed_names is a list of [la
keyed_names = []
for name in names:
  keyed_names.append([last_na
```

```
keyed_names: [['Newton', 'Isaac Newton'], ['Newton', 'Fred Newton'],
['Bohr', 'Niels Bohr']]

sorted(keyed_names): [['Bohr', 'Niels Bohr'], ['Newton', 'Fred Newton'],
['Newton', 'Isaac Newton']]

sorted(keyed_names, reverse = True): [['Newton', 'Isaac Newton'],
['Newton', 'Fred Newton'], ['Bohr', 'Niels Bohr']]
```

Take a look at the list you created, it can no

```
print("keyed_names:", keyed_names)
print("sorted(keyed_names):", sorted(keyed_names))
print("sorted(keyed_names, reverse = True):")
print(sorted(keyed_names, reverse
```

```
sorted_names: ['Isaac Newton', 'Fred Newton', 'Niels Bohr']
```

(This works because Python compares two elements that are lists *elementwise*.)

```
sorted_keyed_names = sorted(keyed_names, reverse = True)
sorted_names = []
for keyed_name in sorted_keyed_names:
  sorted_names.append(keyed_name[1])
print("sorted_names:", sorted_names)
```

2) Sort the list new list.

3) Extract the relevant part.

2. Use a sort key as the **key** argument

Supply the **key** argument to the **sorted** function or the **sort** function

```
def last_name(str):
    return str.split(" ")[1]

names = ["Isaac Newton", "Fred Newton", "Niels Bohr"]
print("sorted(names, key = last_name):")
print(sorted(names, key = last_name))

print("sorted(names, key = last_name, reverse = True):")
print(sorted(names, key = last_name, reverse = True))

print(sorted(names, key =

def last_name_len(name):
    return len(last_name(n

print(sorted(names, key =
```

```
sorted(names, key = last_name): ['Niels Bohr',
'Isaac Newton', 'Fred Newton']

sorted(names, key = last_name, reverse = True):
['Isaac Newton', 'Fred Newton', 'Niels Bohr']

['Niels Bohr', 'Fred Newton', 'Isaac Newton']
['Niels Bohr', 'Isaac Newton', 'Fred Newton']
```

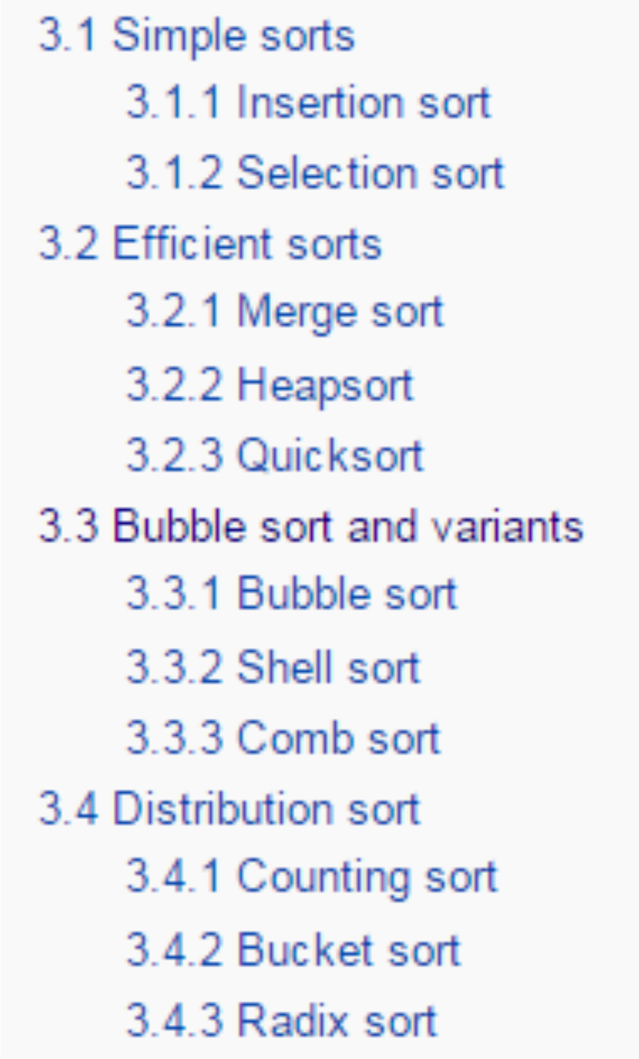
Sorting: strings vs. numbers

- Sorting the powers of 5:

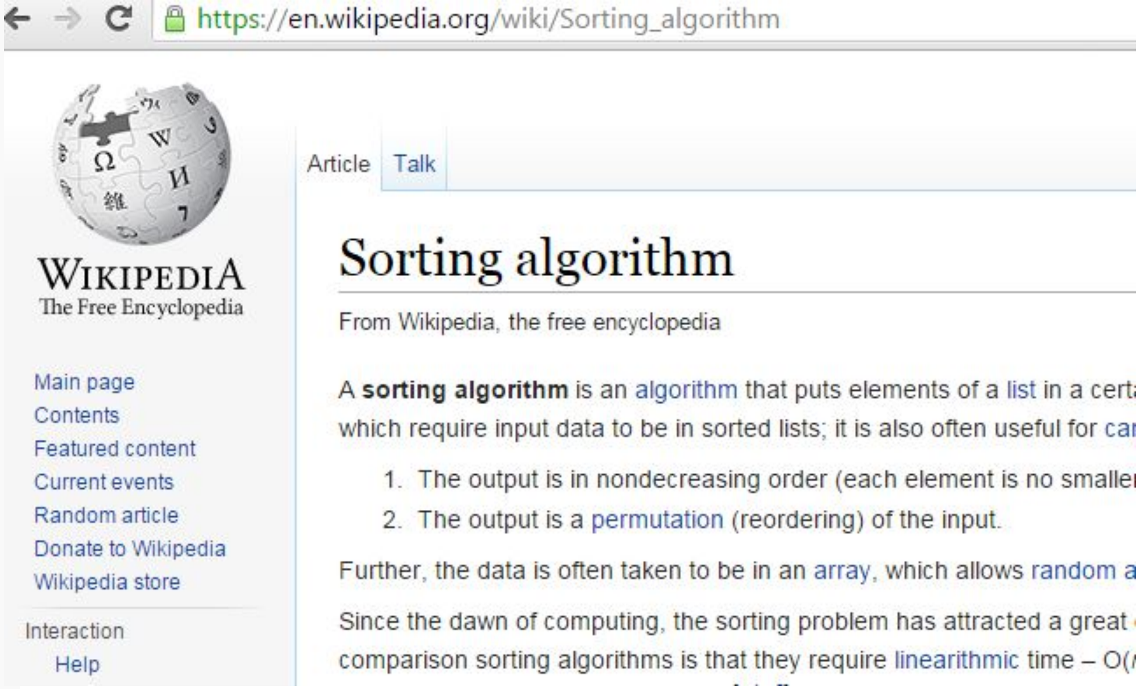
```
>>> sorted([125, 5, 3125, 625, 25])
[5, 25, 125, 625, 3125]

>>> sorted(["125", "5", "3125", "625", "25"])
['125', '25', '3125', '5', '625']
```

Sorting Algorithms Revisited



- 3.1 Simple sorts
  - 3.1.1 Insertion sort
  - 3.1.2 Selection sort
- 3.2 Efficient sorts
  - 3.2.1 Merge sort
  - 3.2.2 Heapsort
  - 3.2.3 Quicksort
- 3.3 Bubble sort and variants
  - 3.3.1 Bubble sort
  - 3.3.2 Shell sort
  - 3.3.3 Comb sort
- 3.4 Distribution sort
  - 3.4.1 Counting sort
  - 3.4.2 Bucket sort
  - 3.4.3 Radix sort



Sorting algorithm

From Wikipedia, the free encyclopedia

A **sorting algorithm** is an algorithm that puts elements of a list in a cert
which require input data to be in sorted lists; it is also often useful for ca

1. The output is in nondecreasing order (each element is no smalle
2. The output is a permutation (reordering) of the input.

Further, the data is often taken to be in an array, which allows random a

Since the dawn of computing, the sorting problem has attracted a great
comparison sorting algorithms is that they require linearithmic time – O(

Bubble Sort

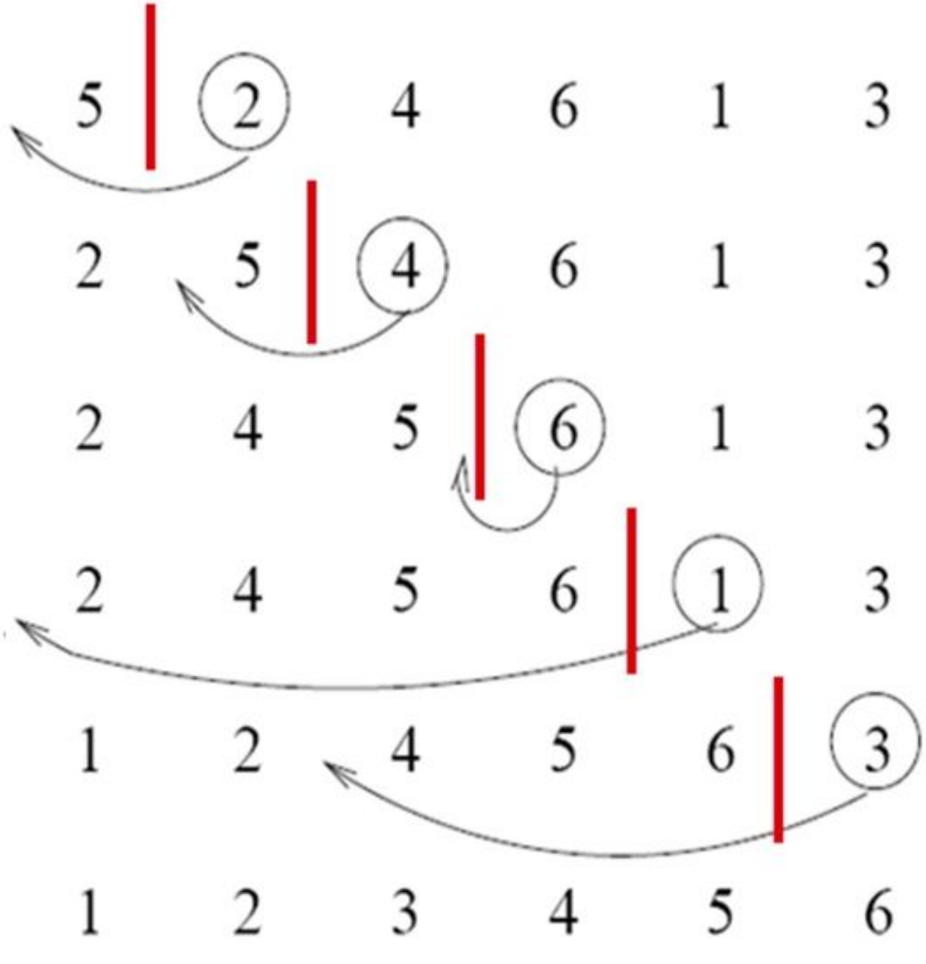
- It repeatedly steps through the list to be sorted,
- compares each pair of adjacent items and swaps them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.

Bubble sort

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Insertion sort



5 2 4 6 1 3

2 5 4 6 1 3

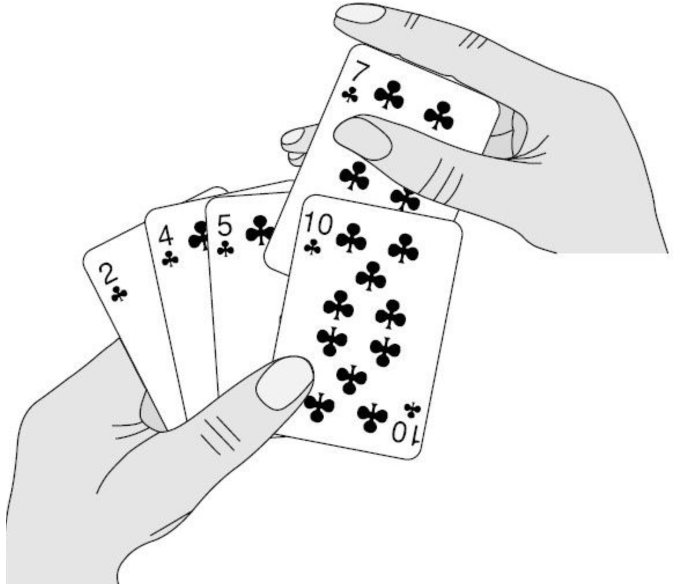
2 4 5 6 1 3

2 4 5 6 1 3

1 2 4 5 6 3

1 2 3 4 5 6

Done !



- maintain a sorted sublist in the lower positions of the list.
- Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger.

Insertion Sort

```
def insertionSort(alist):
    for index in range(1,len(alist)):
        currentvalue = alist[index]
        position = index

        while position>0 and alist[position-1]>currentvalue:
              alist[position]=alist[position-1]
              position = position-1

        alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]
insertionSort(alist)
print(alist)
```

Merge Sort

- Merge sort is a prototypical divide-and-conquer algorithm.

- It was invented in 1945, by John von Neumann.

- Like many divide-and-conquer algorithms it is most easily described recursively.
  1. If the list is of length 0 or 1, it is already sorted.
  2. If the list has more than one element, split the list into two lists, and use mergesort to sort each of them.
  3. Merge the results.

Merge Sort

```
def merge(left, right):
    result = []
    (i,j) = (0, 0)

    while i<len(left) and j<len(right):
       if left[i]<right[j]:
          result.append(left[i])
          i = i + 1
       else:
          result.append(right[j])
          j = j + 1

    while i<len(left):
       result.append(left[i])
       i = i + 1

    while j<len(right):
       result.append(right[j])
       j = j + 1

    return result
```

Merge Sort

Visit this slide later when we learned about recursion.

```
def mergeSort(L):
    if len(L)<2:
        return L[:]
    else:
        middle = len(L)//2
        left = mergeSort(L[:middle])
        right = mergeSort(L[middle:])
        return merge(left, right)

a = mergeSort([2,1,3,4,5,-1,8,6,7])
```

Three Ways to Define a List

- Explicitly write out the whole thing:

```
squares = [0, 1, 4, 9, 16, 25, 36, 49]
```

- Write a loop to create it:

```
squares = []
for i in range(8):
    squares.append(i*i)
```

- Write a **list comprehension**:

```
squares = [i*i for i in range(8)]
```

A list comprehension is a concise description of a list
A list comprehension is shorthand for a loop

Two ways to convert Centigrade to Fahrenheit

```
ctemps = [17.1, 22.3, 18.4, 19.1]
```

With a loop:

```
ftemps = []
for c in ctemps:
  f = celsius_to_farenheit(c)
  ftemps.append(f)
```

With a list comprehension:

```
ftemps = [celsius_to_farenheit(c) for c in ctemps]
```

The comprehension is usually shorter, more readable, and more efficient.

Syntax of a Comprehension

```
[(x,y) for x in seq1 for y in seq2 if sim(x,y) > threshold]
```

expression

for clause (required) assigns value to the variable x

zero or more additional **for** clauses

zero or more **if** clauses

something that can be iterated

Semantics of a comprehension

```
[(x,y) for x in seq1 for y in seq2 if sim(x,y) > threshold]
```

```
result = []
for x in seq1:
  for y in seq2:
    if sim(x,y) > threshold:
      result.append( (x,y) )
… use result …
```

Types of comprehensions

List

```
[ i*2 for i in range(3) ]
```

Set

```
{ i*2 for i in range(3)}
```

Dictionary

{ *key*: *value* for *item* in *sequence* …}

```
{ i: i*2 for i in range(3)}
```

Cubes of the first 10 natural numbers

Goal:

Produce: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

With a loop:

```
cubes = []
for x in range(10):
  cubes.append(x**3)
```

With a list comprehension:

```
cubes = [x**3 for x in range(10)]
```

Powers of 2, 2^0 through 2^{10}

Goal: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

```
[2**i for i in range(11)]
```

Even elements of a list

Goal: Given an input list **`nums`**, produce a list of the even numbers in **`nums`**

```
nums = [3, 1, 4, 1, 5, 9, 2, 6, 5]
```

$\Rightarrow$ [4, 2, 6]

```
[num for num in nums if num % 2 == 0]
```

Dice Rolls

Goal: A list of all possible dice rolls.

With a loop:

```
rolls = []
for r1 in range(1,7):
  for r2 in range(1,7):
     rolls.append((r1,r2))
```

With a list comprehension:

```
rolls = [ (r1,r2) for r1 in range(1,7)
                       for r2 in range(1,7)]
```

All above-average 2-die rolls

Goal: Result list should be a list of 2-tuples:

```
[(2, 6), (3, 5), (3, 6), (4, 4), (4, 5), (4, 6), (5, 3), (5, 4),
(5, 5), (5, 6), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6)]
```

```
[(r1, r2) for r1 in [1,2,3,4,5,6]
              for r2 in [1,2,3,4,5,6]
                 if r1 + r2 > 7]
```

OR

```
[(r1, r2) for r1 in range(1, 7)
              for r2 in range(8-r1, 7)]
```

Making a Matrix

Goal: A matrix were each element is the sum of it's row and column numbers.

With a loop:

```
matrix = []
for i in range(5):
    row = []
    for j in range(5):
        row.append(i+j)
    matrix.append(row)
```

```
[[0, 1, 2, 3, 4],
 [1, 2, 3, 4, 5],
 [2, 3, 4, 5, 6],
 [3, 4, 5, 6, 7],
 [4, 5, 6, 7, 8]]
```

With a list comprehension:

```
matrix = [[i+j for j in range(5)] for i in range(5)]
```

Function $4x^2 – 4$

With a loop:

```
num_list = []
for i in range(-10,11):
    num_list.append(4*i**2 - 4)
```

With a list comprehension:

```
num_list = [4*i**2 - 4 for i in range(-10,11)]
```

Normalize a List

With a loop:

```
num_list = [6,4,2,8,9,10,3,2,1,3]
total = float(sum(num_list))
for i in range(len(num_list)):
    num_list[i] = num_list[i]/float(total)
```

With a list comprehension:

```
num_list = [i/total for i in num_list]
```

Dictionary Mapping Integers to Multiples Under 20

With a loop:

```
for n in range(1,11):
    multiples_list = []
    for i in  range(1,21):
        if i%n == 0:
            multiples_list.append(i)
    multiples[n] = multiples_list
```

With a dictionary comprehension:

```
multiples = {n:[i for i in range(1,21) if i%n == 0]
for n in range(1,11) }
```

```
{1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20], 2: [2, 4, 6, 8, 10, 12, 14,
16, 18, 20], 3: [3, 6, 9, 12, 15, 18], 4: [4, 8, 12,
16, 20], 5: [5, 10, 15, 20], 6: [6, 12, 18], 7: [7,
14], 8: [8, 16], 9: [9, 18], 10: [10, 20]}
```

A Word of Caution

List comprehensions are great, but they can get confusing. Error on the side of readability.

```
nums = [n for n in range(100) if
        sum([int(j) for j in str(n)]) % 7 == 0]

nums = []
for n in range(100):
   digit_sum = sum([int(j) for j in str(n)])
   if digit_sum % 7 == 0:
      nums.append(n)
```

Ternary Assignment

A common pattern in python

```
if x > threshold:
    flag = True
else:
  flag = False
```

Or

```
flag = False
if x > threshold:
  flag = True
```

Ternary Assignment

A common pattern in python

```
if x > threshold:
    flag = True
else:
  flag = False
```

```
flag = True if x > threshold else False
```

Ternary Expression
Three elements

Ternary Assignment

$$\texttt{flag = }\underbrace{\texttt{True}}_{\text{Result if true}}\ \mathtt{\textbf{if}}\ \underbrace{\texttt{x > threshold}}_{\text{Condition}}\ \mathtt{\textbf{else}}\ \underbrace{\texttt{False}}_{\text{Result if false}}$$

- Only works for single expressions as results.
- Only works for if and else (no elif)

Ternary Assignment

Goal: A list of 'odd' or 'even' if that index is odd or even.

```
the_list = []
for i in range(16):
    if i%2 == 0:
        the_list.append('even')
    else:
        the_list.append('odd')
```

or

```
the_list = []
for i in range(16):
    the_list.append('even' if i%2 == 0 else 'odd')
```

Ternary Assignment

Goal: A list of 'odd' or 'even' if that index is odd or even.

```
the_list = []
for i in range(16):
    if i%2 == 0:
        the_list.append('even')
    else:
        the_list.append('odd')
```

or

```
the_list =
   ['even' if i%2 == 0 else 'odd' for i in range(16)]
```

Lecture Overview

- Arrays
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

Tuples

- Like strings, **tuples** are ordered sequences of elements.
- The individual elements can be of any type, and need not be of the same type as each other.
- Literals of type tuple are written by enclosing a comma-separated list of elements within parentheses.
- Tuples differ from lists in one hugely important way:
  - Lists are mutable. In contrast, tuples are immutable.

-
```
t1 = ()
t2 = (1, 'two', 3)
print(t1)
print(t2)

>> ()
>> (1, 'two', 3)
```

Tuples

- Like strings, tuples can be concatenated, indexed, and sliced.

-
  ```
  t1 = (1, 'two', 3)
  t2 = (t1, 3.25)
  print(t2)
  print((t1 + t2))
  print((t1 + t2)[3])
  print((t1 + t2)[2:5])

  >> ((1, 'two', 3), 3.25)
  >> (1, 'two', 3, (1, 'two', 3), 3.25)
  >> (1, 'two', 3)
  >> (3, (1, 'two', 3), 3.25)
  ```

Tuples

- A for statement can be used to iterate over the elements of a tuple.
- The following code prints the common divisors of 20 and 100 and then the sum of all the divisors.

-
  ```
def findDivisors (n1, n2):
    """Assumes n1 and n2 are positive ints
       Returns a tuple containing all common divisors
       of n1 & n2"""
    divisors = () #the empty tuple
    for i in range(1, min (n1, n2) + 1):
       if n1%i == 0 and n2%i == 0:
          divisors = divisors + (i,)
    return divisors

divisors = findDivisors(20, 100)
print(divisors)
total = 0
for d in divisors:
    total += d
print(total)
```

```
>> (1, 2, 4, 5, 10, 20)
>> 42
```