Algorithmic Thinking: Loops and Conditionals

Last Time

- A control flow structure: `for` loop
- ```
  range(n)
  range(start, end)
  range(start, end, step)
  ```
- Assignments that modify variables:

  ```
  x = x + y
  ```
  ```

Iteration with for loops

```
def test1():
    for i in range(1,6):
        print("Woof")
```

```
>>> test1()
Woof
Woof
Woof
Woof
Woof
```

What determines how many times "Woof" is printed is **the number of elements in the range.**

Any expression that gives 5 elements in the range would give the same output.

For example,
range(5), range(0,5), …

Iteration with for loops

```
def test2():
    for i in range(1,6):
        print(i, end='-')
```

```
>>> test2()
1-2-3-4-5-
```

```
range(5)          →?
range(0, 5)       →?
range(1, 6)       →?

range(1, 10, 2)   →?
range(2, 10, 2)   →?

range(10, 1, -1)  →?
range(10, 2, -4)  →?
```

Iteration with for loops

```
def test3():
    for i in range(1,6):
        print("Woof" * i)
```

This expression creates a string that concatenates *i* number of "Woof"s.

```
>>> test3()
Woof
WoofWoof
WoofWoofWoof
WoofWoofWoofWoof
WoofWoofWoofWoofWoof
```

This Lecture

- The notion of an algorithm
- Moving from algorithm to code
- Python control structures:
  - While loops, conditionals

Algorithms

- An algorithm is "a **precise rule (or set of rules)** specifying **how to solve some problem**." (thefreedictionary.com)

- The **study of algorithms** is one of the foundations of computer science.

Mohammed al-Khowarizmi (äl-khōwärēz´mē)

Persian mathematician of the court of Mamun in Baghdad…
the word ***algorithm*** is said to have been derived from his name.
Much of the mathematical knowledge of medieval Europe was derived from Latin translations of his works. (encyclopedia.com)

An algorithm is like a function

$$F(x) = y$$

Suggestion: use paper and pen before keyboard

Input

- **Input specification**
  - Recipes: ingredients, cooking utensils, …
  - Knitting: size of garment, length of yarn, needles …
  - Tax Code: wages, interest, tax withheld, …

- Input specification for computational algorithms:
  - **What kind of data** is required?
  - **In what form** will this data be received by the algorithm?

Computation

- An algorithm requires **clear and precisely stated steps** that express how to perform the operations to yield the desired results.

- Algorithms assume a basic set of ***primitive operations*** that are assumed to be understood by the executor of the algorithm.
  - Recipes: beat, stir, blend, bake, …
  - Knitting: casting on, slip loop, draw yarn through, ...
  - Tax code: deduct, look up, check box, …
  - Computational: add, set, modulo, output, …

Output

- **Output specification**
  - Recipes: number of servings, how to serve
  - Knitting: final garment shape
  - Tax Code: tax due or tax refund, where to pay

- Output specification for computational algorithms:
  - **What results are required?**
  - **How should these results be reported?**
  - **What happens if no results can be computed** due to an error in the input? What do we output to indicate this?

Is this a "good" algorithm?

Input: slices of bread, jar of peanut butter, jar of jelly

1. **Pick up some bread.**
2. **Put peanut butter on the bread.**
3. **Pick up some more bread.**
4. **Open the jar of jelly.**
5. **Spread the jelly on the bread.**
6. **Put the bread together to make your sandwich.**

Output?

What makes a "good" algorithm?

- A good algorithm should produce the **correct outputs for any set of legal inputs.**
- A good algorithm should **execute efficiently with the fewest number of steps as possible.**
- A good algorithm should be designed in such a way that **others will be able to understand it and modify it to specify solutions to additional problems.**

An epidemic (covered last week)

```
def compute_sick(numOfDays):
    #computes total sick after n days
    newly_sick = 1  #initially 1 sick person
    total_sick = 1

    for day in range(2, numOfDays + 1):
    #each iteration represents one day
        newly_sick = newly_sick * 2
        total_sick = total_sick + newly_sick

    return total_sick
```

Each newly infected person infects 2 people the next day.
The function returns the number of sick people after n days.

Variation on the Epidemic Example

Let us write a function that

- **Inputs the size of the population**
- **Outputs the number of days left before all the population dies out**

How can we do that using iteration (loops)?

Keep track of the number of sick people.

But do we know how many times we should loop?

Recall the Epidemic Example

```
def days_left(population):
    #computes the number of days until extinction
    days = 1
    newly_sick = 1
    total_sick = 1
    while total_sick < population:
        #each iteration represents one day
        newly_sick = newly_sick * 2
        total_sick = total_sick + newly_sick
        days = days + 1
    print(days, " days for the population to die off")
    return days
```

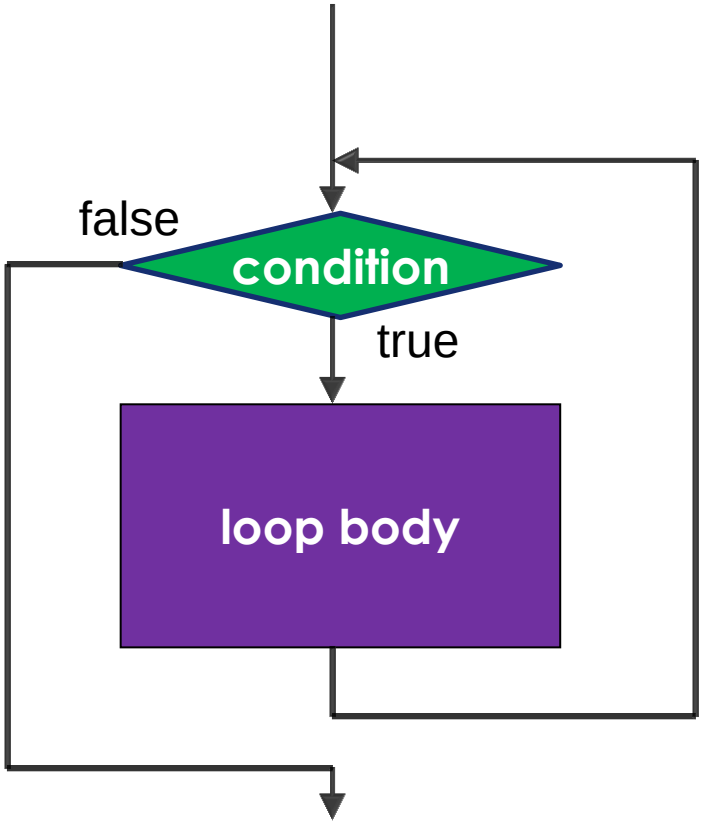
`while` loop

Format:

```
while condition:
    loop body
    loop body
```

one or more instructions
to be repeated

After the loop condition becomes false during the loop body, the loop body still runs to completion (before its check before the next turn) and exit the loop and go on with the next step.



Recall the Epidemic Example

```
def days_left(population):
    #computes the number of days until extinction
    days = 1
    newly_sick = 1
    total_sick = 1
    while total_sick < population:
        #each iteration represents one day
        newly_sick = newly_sick * 2
        total_sick = total_sick + newly_sick
        days = days + 1
    print(days, " days for the population to die off")
    return days
```

Loop condition

Should be changing so that loop will end at a point

While Loop Examples

What is the output?

```
i = 1
while  i < 6:
    print(i, end=' ')
    i = i + 1
print('\n After :', i)
```

'**\n**' *means* ***new line***

How about this?

```
i = 0
while  i < 5:
    i = i + 1
    print(i , end=' ')
print('\n After :', i)
```

What is the value of i when we exit the loop?

While Loop Examples

```
i = 1
while  i < 6:
    print(i, end=' ')
    i = i + 1
print('\n', 'After :',i)
print('-------------');
i = 0
while  i < 5:
    i = i + 1
    print(i , end=' ')
print('\n After :', i)
```

```
>>>
1 2 3 4 5
 After : 6
-------------
1 2 3 4 5
 After : 5
>>>
```

While vs. For Loops

Prints first 10 positive integers

```
i = 1
while  i < 11:
    print(i)
    i = i + 1
```

Prints first 10 positive integers

```
for i in range(1,11):
  print(i)
```

When to use `for` or `while` loops

- If you know in advance **how many times** you want to run a loop use a `for` loop.

- **When you don't know the number** of repetition needed, use a `while` loop.

A Simple Algorithm

Input numerical ***score*** between 0 and 100 and
Output "Pass" or "Fail"

Algorithm:

1. **If *score* >= 60**
    a. Set ***grade*** to "Pass"
    b. Print "Pass"
2. **Otherwise**,
    a. Set ***grade*** to "Fail"
    b. Print "Fail"
3. Print "See you in class"
4. Return ***grade***

Exactly one of the steps 1 or 2 is executed, but step 3 and step 4 are always executed.

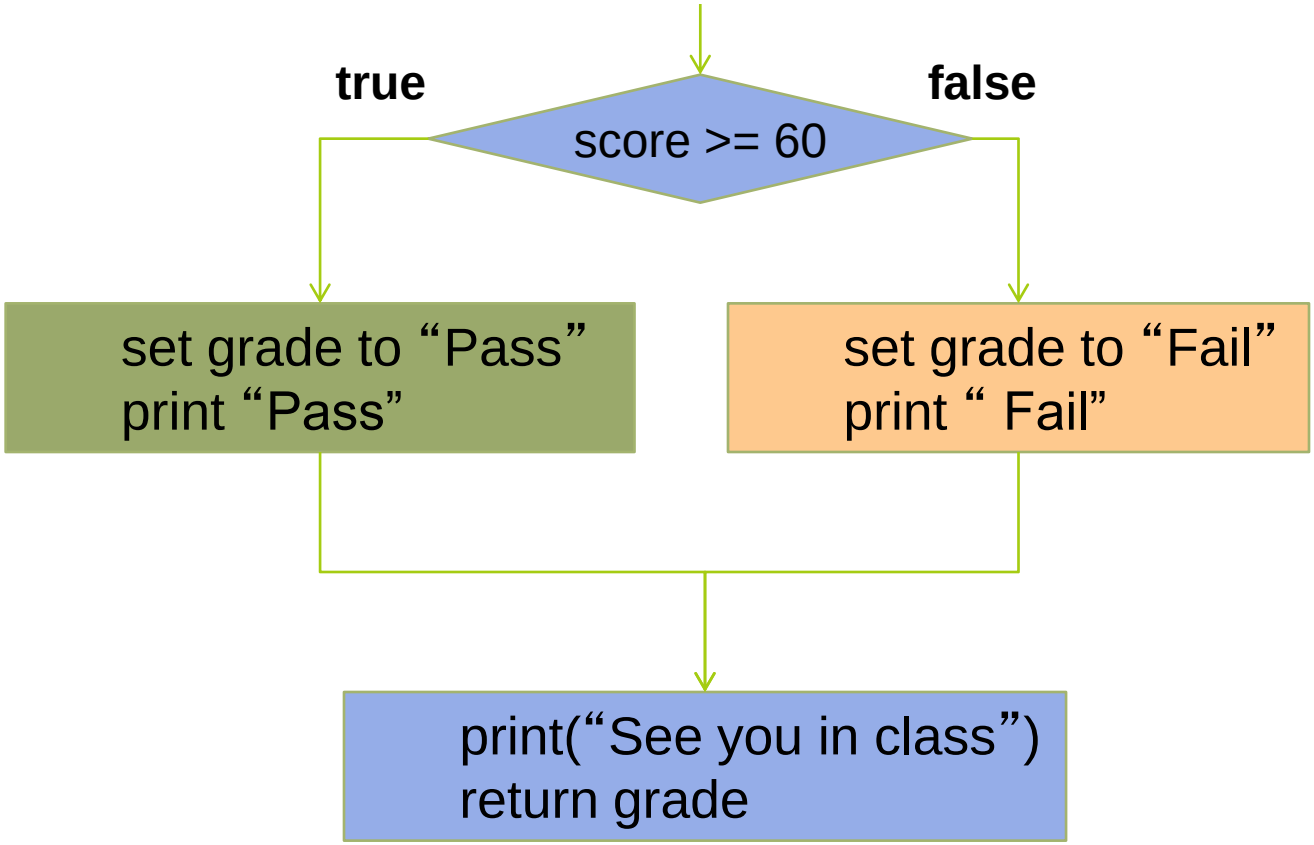
Coding the Grader in Python

Algorithm:

1. If *score* >= 60
    a. Set *grade* to “Pass”
    b. Print “ Pass”
2. Otherwise,
    a. Set *grade* to “Fail”
    b. Print “Fai”
3. Print “See you in class ”
4. Return *grade*

```
def grader(score):
    if score >= 60:
        grade = "Pass"
        print("!!!Pass")
    else:
        grade = "Fail"
        print("!!!Fail")
    print("See you in class")
    return grade
```

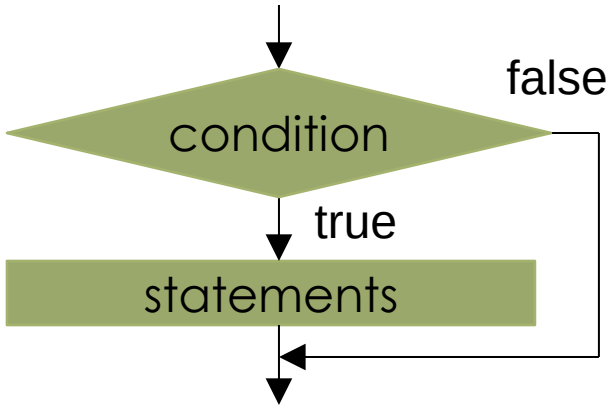
Control Flow



FLOW CHART: `if` statement

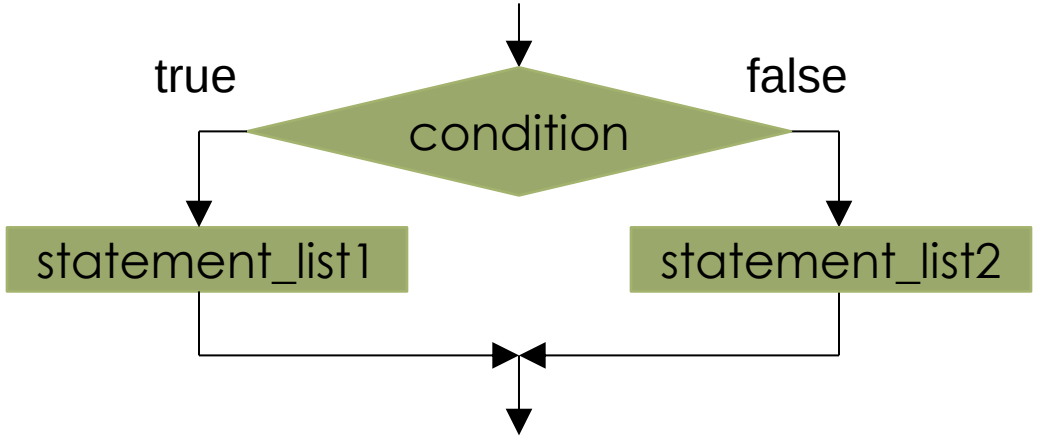
Format:

```
if condition :
    statement_list
```



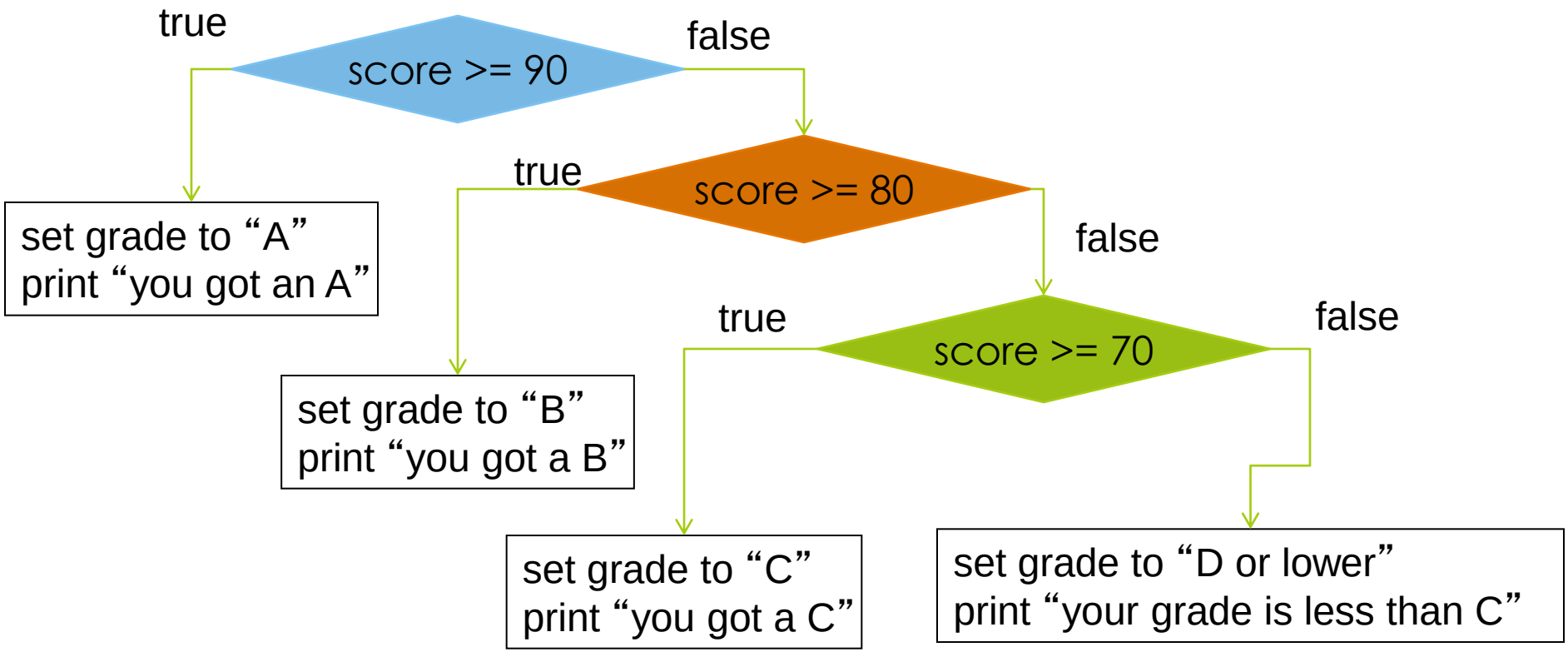
FLOW CHART: `if/else` statement

Format:




```
if condition :
      statement_list1
else:
      statement_list2
```

Grader for Letter Grades



Nested if statements

```
def grader2(score):
    if score >= 90:
        grade = "A"
        print("You got an A")
    else: # score less than 90
        if score >= 80:
            grade = "B"
            print("You got a B")
        else: # score less than 80
            if score >= 70:
                grade = "C"
                print("You got a C")
            else: #score less than 70
                grade = "D or lower"
                print("Your grade is less than C")
    return grade
```

Equivalently

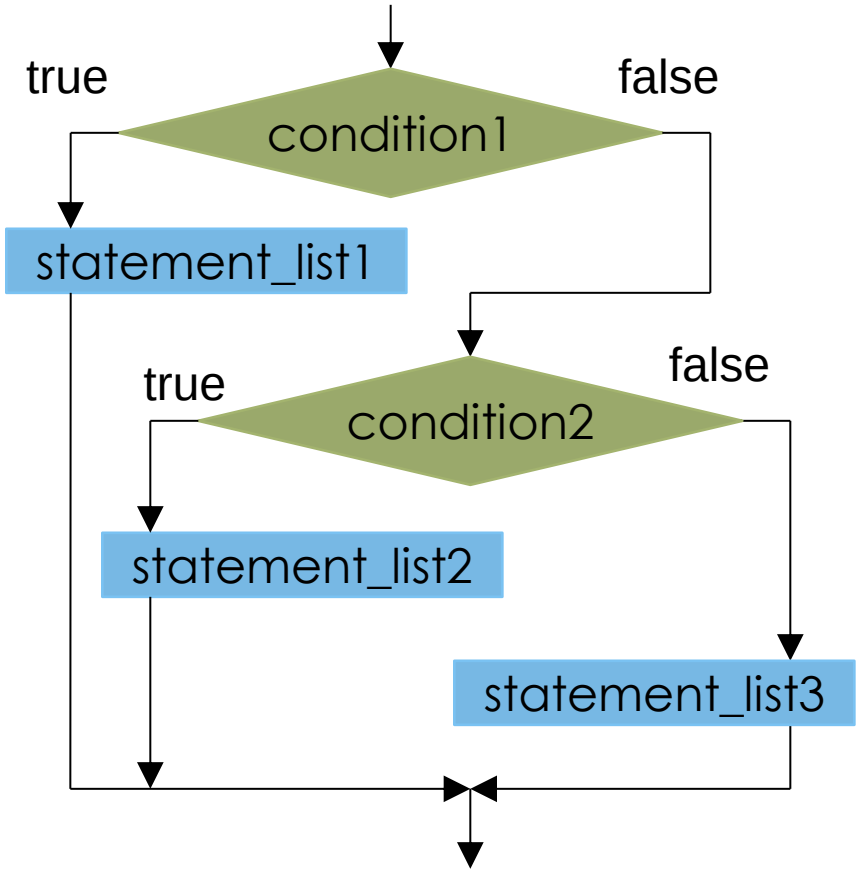
```
def grader3(score):
    if score >= 90:
        grade = "A"
        print("You got an A")
    elif score >= 80:
        grade = "B"
        print("You got a B")
    elif score >= 70:
        grade = "C"
        print("You got a C")
    else:
        grade = "D or lower"
        print("Your grade is less than C")
    return grade
```

```
if score >= 90:
    grade = "A"
    print("You got an A")
else: # score less than 90
    if score >= 80:
        grade = "B"
        print("You got a B")
    else: # score less than 80
        if score >= 70:
```

Flow chart: `if/elif/else` statement

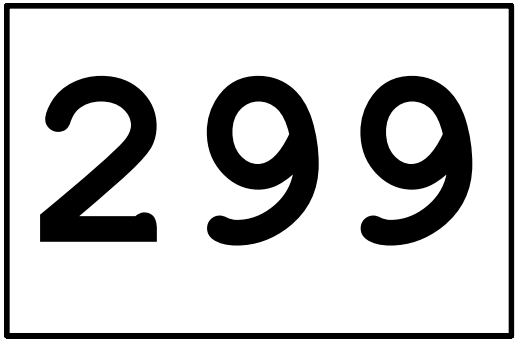
Format:

```
if condition1:
    statement_list1
elif condition2 :
    statement_list2
else:
    statement_list3
```



Example: Finding the maximum

How do we find the maximum in a sequence of integers shown to us one at a time?

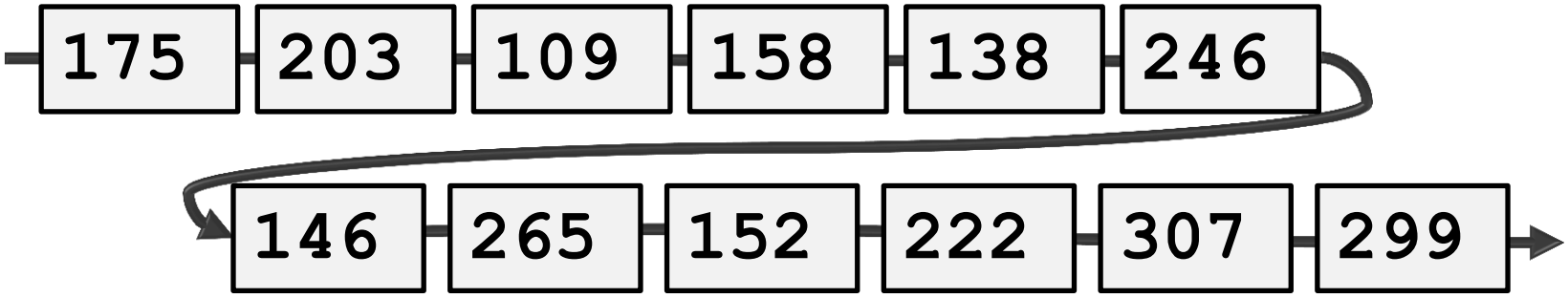


What's the maximum?

This slide is added just for the lecture notes in PDF.
Same with the previous slide which animates the list of numbers

Example: Finding the maximum

How do we find the maximum in a sequence of integers shown to us one at a time?



What's the maximum?

Example: Finding the maximum

Input: a non-empty ***list*** of integers.

1. Set ***max_so_far*** to the first number in ***list***.
2. For each number ***n*** in ***list***:
    a. If ***n*** is greater than ***max_so_far***, then set ***max_so_far*** to ***n***.

Loop

Output: ***max_so_far*** as the maximum of the ***list***.

Until Now

- Notion of an algorithm:
  - Kinds of instructions needed to express algorithms
  - What makes an algorithm a good one
- Instructions for specifying control flow (for loop, while loop, if/then/else)
  - Flow charts to express control flow in a language-independent way
  - Coding these control flow structures in Python

NEXT > Lists!

> Organizing/Processing lots of data.

Representing Lists in Python

We will use a **list** to represent a collection of data values.

```
scores = [78, 93, 80, 68, 100, 94, 85]
colors = [ ‘red’ , ‘green’ , ‘blue’ ]
mixed = [ ‘purple’ , 100, 90.5]
```

A list is an *ordered* sequence of values and may contain values of any data type.

In Python lists may be heterogeneous (may contain items of different data types).

Some List Operations

- **Indexing** (think of subscripts in a sequence)
- **Length** (number of items contained in the list)
- **Slicing**
- **Membership** check
- **Concatenation**
- …

Some List Operations

```
>>> names = [ "Al", "Jane", "Jill", "Mark" ]

>>> len(names)
4

>>> Al in names
Error  ... name 'Al' is not defined

>>> "Al" in names
True

>>> names + names
['Al', 'Jane', 'Jill', 'Mark', 'Al', 'Jane', 'Jill', 'Mark']
```

Accessing List Elements

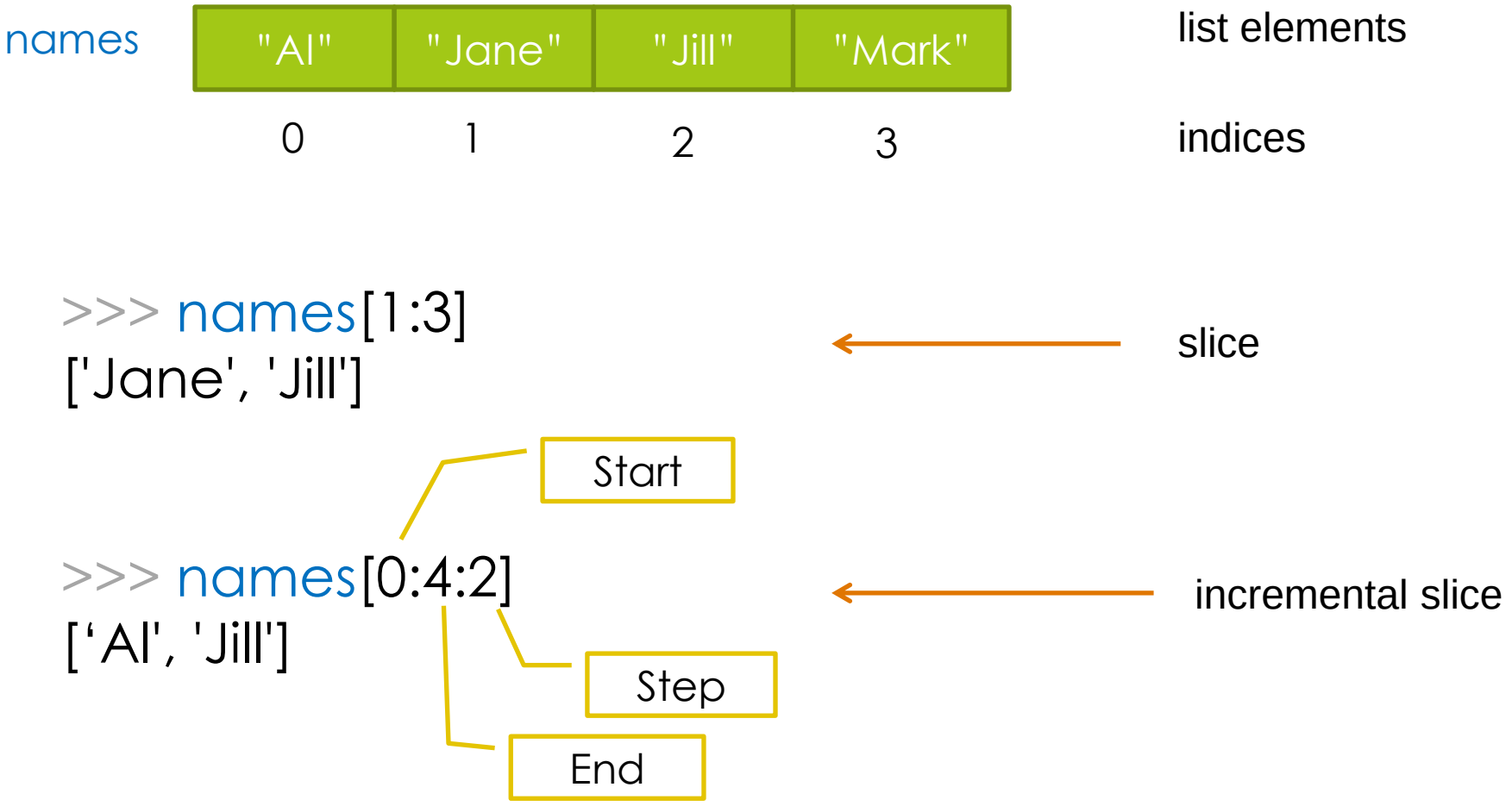

```
>>> names[0]
'Al'
```

```
>>> names[3]
'Mark'
```

```
>>> names[len(names)-1]
'Mark'
```

```
>>> names[4]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    names[4]
IndexError: list index out of range
```

Slicing Lists



Slicing Lists


```
names, names[0:4], names[0,4,1]
They all refer to  ['Al', 'Jane', 'Jill', 'Mark']

>>> names[1:3]       ['Jane', 'Jill']
>>> names[1:4]       ['Jane', 'Jill', 'Mark']

>>> names[0:4:2]     ['Al', 'Jill']
>>> names[:3]        ['Al', 'Jane', 'Jill']

>>> names[:2]        ['Al', 'Jane']
>>> names[2:]        ['Jill', 'Mark']
```

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n, n * s`	*n* shallow copies of *s* concatenated
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(i)`	index of the first occurence of *i* in *s*
`s.count(i)`	total number of occurences of *i* in *s*

source: docs.python.org

Modifying Lists

```
>>> names = ['Al', 'Jane', 'Jill', 'Mark']
>>> names[1] = "Kate"
>>> names
['Al', 'Kate', 'Jill', 'Mark']
```

```
>>> names[1:3] = [ "Me", "You" ]
>>> names
['Al', 'Me', 'You', 'Mark']
```

```
>>> a = [1, 2, 3]
>>> a[0:0] = [-2, -1, 0]
>>> a
[-2, -1, 0, 1, 2, 3]

>>> a = [1, 2, 3]
>>> a[0:1] = [-2, -1, 0]
>>> a
[-2, -1, 0, 2, 3]
```

```
>>> names[1:3] = [ "AA", "BB", "CC", "DD" ]
['Al', 'AA', 'BB', 'CC', 'DD', 'Mark']
```

The list grew in length, we could make it shrink as well.

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	same as `s[len(s):len(s)] = [x]`
`s.extend(x)`	same as `s[len(s):len(s)] = x`
`s.count(x)`	return number of *i*'s for which `s[i] == x`
`s.index(x[, i[, j]])`	return smallest *k* such that `s[k] == x` and `i <= k < j`
`s.insert(i, x)`	same as `s[i:i] = [x]`
`s.pop([i])`	same as `x = s[i]; del s[i]; return x`
`s.remove(x)`	same as `del s[s.index(x)]`
`s.reverse()`	reverses the items of *s* in place
`s.sort([key[, reverse]])`	sort the items of *s* in place

source: docs.python.org

Tomorrow

- We will continue Lists and algorithms