Definite Loops in Python

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College

`for` Loops

- A `for` statement is one way to create a loop in Python.
  - allows us to *repeat* one or more statements.
- Example:

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```

the body of the loop

will output:

```
Warning
1
Warning
2
Warning
3
```

- The repeated statement(s) are known as the *body* of the loop.
  - must be indented the same amount

`for` Loops (cont.)

- General syntax:

```
for variable in sequence:
    body of the loop
```

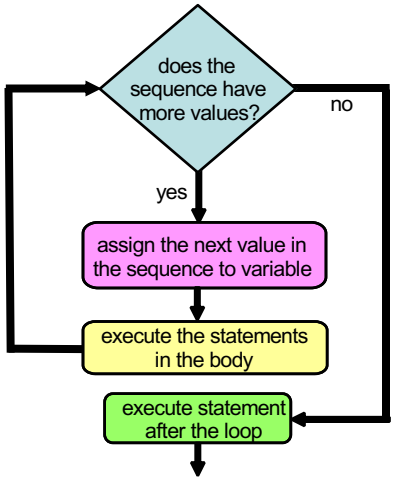
```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```

- For each value in the sequence:
  - the value is assigned to the variable
  - all statements in the body of the loop are executed using that value
- Once all values in the sequence have been processed, the program continues with the first statement after the loop.

Executing a `for` Loop

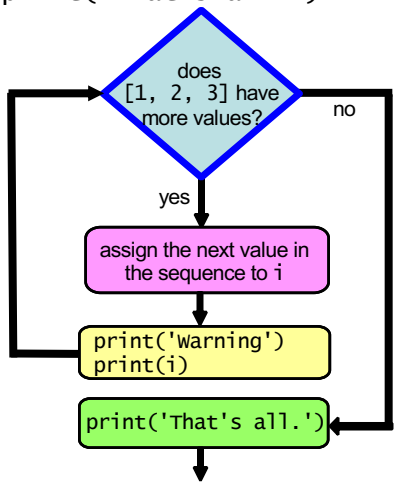
```
for variable in sequence:
    body of the loop
```

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
```



Executing Our Earlier Example
(with one extra statement)

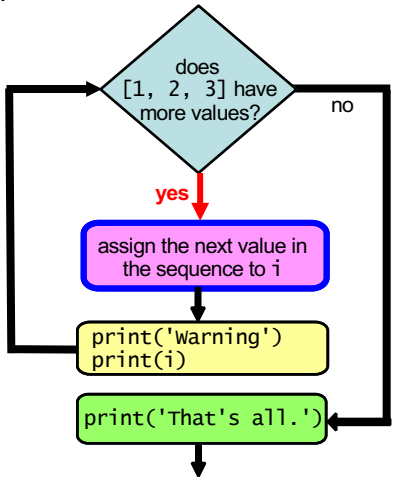
```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action
yes		

Executing Our Earlier Example
(with one extra statement)

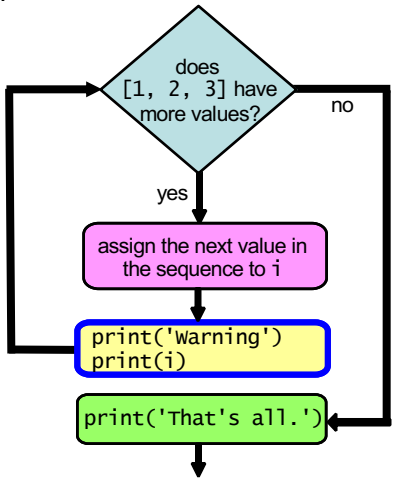
```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action
yes	1	

Executing Our Earlier Example
(with one extra statement)

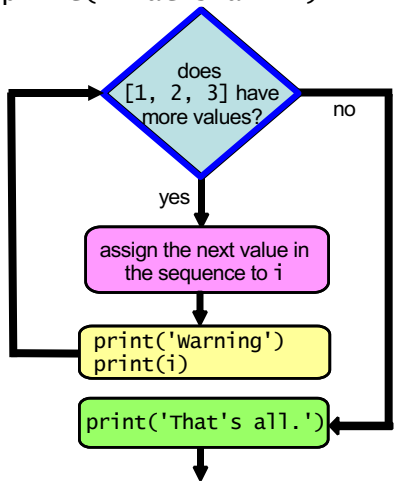
```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action
yes	1	Warning
		1

Executing Our Earlier Example
(with one extra statement)

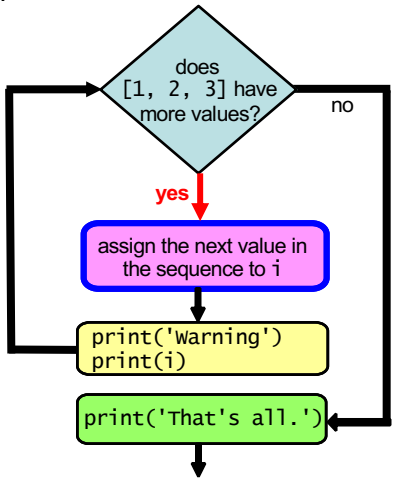
```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action
yes	1	Warning
		1
yes		

Executing Our Earlier Example
(with one extra statement)

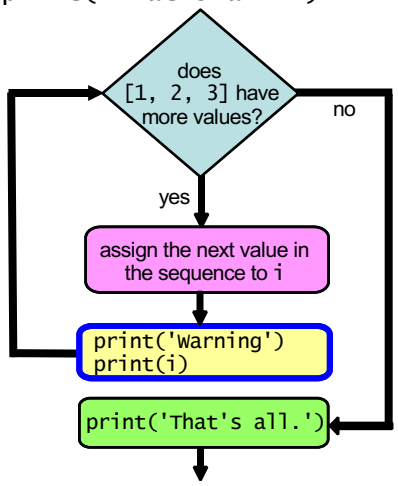
```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action
yes	1	Warning 1
yes	2	

Executing Our Earlier Example
(with one extra statement)

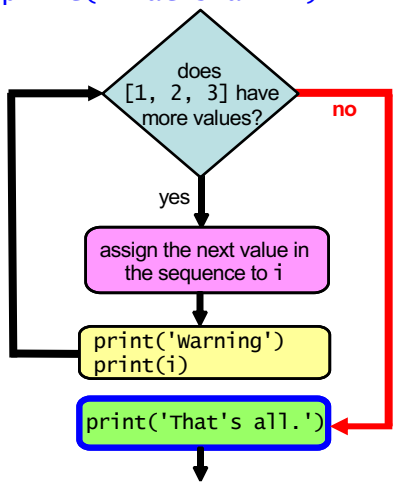
```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action
yes	1	Warning 1
yes	2	Warning 2

Executing Our Earlier Example
(with one extra statement)

```
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```



more?	i	output/action
yes	1	`Warning` `1`
yes	2	`Warning` `2`
yes	3	`Warning` `3`
no		`That's all.`

Another Example

- What would this code output?

```
for val in [2, 4, 6, 8, 10]:
    print(val * 10)
print(val)
```

- Use a table to help you:

more?	val	output/action

Another Example

- What would this code output?

```
for val in [2, 4, 6, 8, 10]:
    print(val * 10)
print(val)
```

- Use a table to help you:

more?	val	output/action
yes	2	20
yes	4	40
yes	6	60
yes	8	80
yes	10	100
no		exit loop
		10

full output:
20
40
60
80
100
10

Simple Repetition Loops

- Recall: `repeat` statements in Scratch allowed us to simply repeat a block of code a specified number of times:

- To get the equivalent of `repeat 10` in Python, we would do this:

```
for i in range(10):
    ...
```

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):      # [0, 1, 2, ..., N-1]
    body of the loop
```

- Example:

```
for i in range(3):       # [0, 1, 2]
    print('I'm feeling loopy!')
```

outputs:

```
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
```

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):      # [0, 1, 2, ..., N-1]
    body of the loop
```

- Example:

```
for i in range(5):       # [0, 1, 2, 3, 4]
    print('I'm feeling loopy!')
```

outputs:

```
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
```

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):          # [0, 1, 2, ..., N-1]
    body of the loop
```

- What would this loop do?

```
for i in range(8):
    print('I'm feeling loopy!')
```

Simple Repetition Loops

- To repeat a loop's body *N* times:

```
for i in range(N):          # [0, 1, 2, ..., N-1]
    body of the loop
```

- What would this loop do?

```
for i in range(8):          # [0, 1, 2, 3, 4, 5, 6, 7]
    print('I'm feeling loopy!')
```

outputs:

```
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
I'm feeling loopy!
```

8 times!

Simple Repetition Loops (cont.)

- Another example:

```
for i in range(7):
    print(i * 5)
```

how many repetitions?

output?

Simple Repetition Loops (cont.)

- Another example:

```
for i in range(7):      # gives [0, 1, 2, 3, 4, 5, 6]
    print(i * 5)
```

how many repetitions? **7**

output?

```
0
5
10
15
20
25
30
```

`for` Loops Are Definite Loops

- *Definite* loop = a loop in which the number of repetitions is *fixed* before the loop even begins.
- In a `for` loop, # of repetitions = `len(`*sequence*`)`

```
for variable in sequence:
    body of the loop
```

To print the warning 20 times,
how could you fill in the blank?

```
for i in ____________________:
    print('Warning!')
```

A. `range(20)`
B. `[1] * 20`
C. `'abcdefghijklmnopqrst'`
D. either A or B would work, but not C
E. A, B or C would work

To print the warning 20 times, how could you fill in the blank?

```
for i in ____________________:
    print('Warning!')
```

A. `range(20)`
B. `[1] * 20`
C. `'abcdefghijklmnopqrst'`

These are all sequences with a length of 20!

D. either A or B would work, but not C
E. **A, B or C would work**

Python Shortcuts

- Consider this code:

```
age = 14
age = age + 1
      14 + 1
        15
```

- Instead of writing

```
age = age + 1
```

we can just write

```
age += 1
```

Python Shortcuts (cont.)

shortcut	equivalent to
var `+=` *expr*	*var* `=` *var* `+` (*expr*)
var `-=` *expr*	*var* `=` *var* `-` (*expr*)
var `*=` *expr*	*var* `=` *var* `*` (*expr*)
var `/=` *expr*	*var* `=` *var* `/` (*expr*)
var `//=` *expr*	*var* `=` *var* `//` (*expr*)
var `%=` *expr*	*var* `=` *var* `%` (*expr*)
var `**=` *expr*	*var* `=` *var* `**` (*expr*)

where *var* is a variable
expr is an expression

- **Important:** the = must come *after* the other operator.
  - `+=` is correct
  - `=+` is not!

To add the numbers in the list `vals`, how could you fill in the blanks?

```
def sum(vals):
    result = 0
    for _____________________________:
        result += _____________________
    return result
```

	first blank	second blank
A.	`x in vals`	`x`
B.	`x in vals`	`vals[x]`
C.	`i in range(len(vals))`	`vals[i]`
D.	either A or B would work, but not C	
E.	either A or C would work, but not B	

To add the numbers in the list `vals`, how could you fill in the blanks?

```
def sum(vals):
    result = 0
    for _____________________________:
        result += _____________________
    return result
```

	first blank	second blank
A.	`x in vals`	`x`
B.	`x in vals`	`vals[x]`
C.	`i in range(len(vals))`	`vals[i]`
D.	either A or B would work, but not C	
E.	**either A or C would work, but not B**	

Using a Loop to Sum a List of Numbers

```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result

print(sum([10, 20, 30, 40, 50]))
```

x	result

Using a Loop to Sum a List of Numbers

```
def sum(vals):                  # vals = [10, 20, 30, 40, 50]
    result = 0
    for x in vals:
        result += x
    return result               # return 150

print(sum([10, 20, 30, 40, 50]))     # print(150)
```

x	result
	0
10	10
20	30
30	60
40	100
50	150

no more values in `vals`, so we're done
output: 150

Cumulative Computations

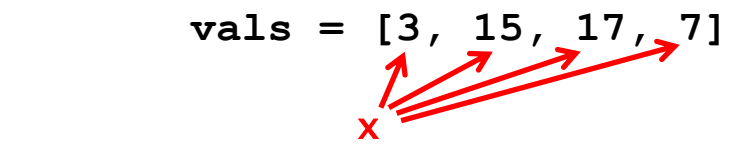
```
def sum(vals):
    result = 0          # the accumulator variable
    for x in vals:
        result += x     # gradually accumulates the sum
    return result

print(sum([10, 20, 30, 40, 50]))
```

x	result
	0
10	10
20	30
30	60
40	100
50	150

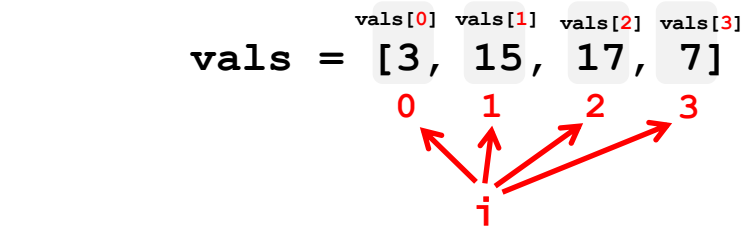
no more values in `vals`, so we're done
output: 150

Element-Based for Loop




```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

Index-Based for Loop




```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

Tracing an Index-Based Cumulative Sum

```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result

print(sum([10, 20, 30, 40, 50]))
```

i	vals[i]	result

Tracing an Index-Based Cumulative Sum

```
def sum(vals):                  # vals = [10, 20, 30, 40, 50]
    result = 0
    for i in range(len(vals)): # range(5) → 0,1,2,3,4
        result += vals[i]
    return result               # return 150

print(sum([10, 20, 30, 40, 50]))  # print(150)
```

i	vals[i]	result
		0
0	10	10
1	20	30
2	30	60
3	40	100
4	50	150

no more values in `range(5)`, so we're done

output: 150

What is the output of this program?

```
def mystery(vals):
    result = 0
    for i in range(len(vals)):
        if vals[i] == vals[i - 1]:
            result += 1
    return result

print(mystery([5, 7, 7, 2, 3, 3, 5]))
```

A. 0
B. 1
C. 2
D. 3
E. 7

i	vals[i]	vals[i - 1]	result

What is the output of this program?

```
def mystery(vals):  # vals = [5, 7, 7, 2, 6, 6, 5]
    result = 0
    for i in range(len(vals)): # range(7) → 0,1,2,3,4,5,6
        if vals[i] == vals[i - 1]:
            result += 1
    return result   # return 3

print(mystery([5, 7, 7, 2, 6, 6, 5]))  # print(3)
```

A. 0
B. 1
C. 2
D. 3
E. 7

i	vals[i]	vals[i - 1]	result
			0
0	5	5	1
1	7	5	1
2	7	7	2
3	2	7	2
4	6	2	2
5	6	6	3
6	5	6	3

output: 3

Element-Based or Index-Based Loop?

```
def mystery(vals):
    result = 0
    for i in range(len(vals)):
        if vals[i] == vals[i-1]:
            result += 1
    return result

print(mystery([5, 7, 7, 2, 6, 6, 5]))
```

Element-Based or *Index-Based* Loop?

```
def mystery(vals):
    result = 0
    for i in range(len(vals)):
        if vals[i] == vals[i-1]:
            result += 1
    return result

print(mystery([5, 7, 7, 2, 6, 6, 5]))
```

Using an *Index-Based* Loop

```
def mystery(vals):
    result = 0
    for i in range(len(vals)):
        if vals[i] == vals[i-1]:
            result += 1
    return result

print(mystery([5, 7, 7, 2, 6, 6, 5]))
```

- What does this program do in general?

- Could we easily do this with an element-based loop?

Using an Index-Based Loop

```
def mystery(vals):
    result = 0
    for i in range(len(vals)):
        if vals[i] == vals[i-1]:
            result += 1
    return result

print(mystery([5, 7, 7, 2, 6, 6, 5]))
```

- What does this program do in general?
  counts the pairs of "adjacent" values that are the same
  (where "adjacent" includes the first and last values)
- Could we easily do this with an element-based loop?
  no! having the index `i` is what allows us to get the pairs of values
  (both `vals[i]` and `vals[i-1]`)

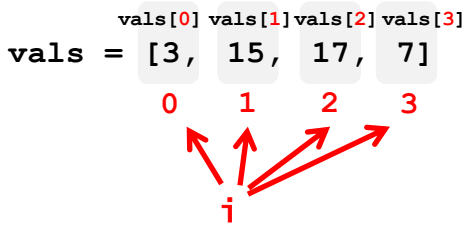
Simpler

```
vals = [3, 15, 17, 7]
```


```
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
```

element-based loop

More Flexible




```
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result
```

index-based loop

More on Cumulative Computations

- We've been performing cumulative computations in assembly — including the loop-based factorial.
- Here's a loop-based factorial in Python:

```
def fac(n):
    result = 1
    for x in range(___________):  # fill in the blank
        result *= x
    return result
```

More on Cumulative Computations

- We've been performing cumulative computations in assembly — including the loop-based factorial.
- Here's a loop-based factorial in Python:

```
def fac(n):
    result = 1
    for x in range(1, n + 1):
        result *= x
    return result
```

More on Cumulative Computations

- We've been performing cumulative computations in assembly — including the loop-based factorial.
- Here's a loop-based factorial in Python:

```
def fac(n):
    result = 1          # the accumulator variable
    for x in range(1, n + 1):
        result *= x     # accumulates the factorial
    return result
```

- Is this loop element-based or index-based?

More on Cumulative Computations

- We've been performing cumulative computations in assembly — including the loop-based factorial.
- Here's a loop-based factorial in Python:

```
def fac(n):
    result = 1          # the accumulator variable
    for x in range(1, n + 1):
        result *= x     # accumulates the factorial
    return result
```

- Is this loop element-based or index-based?
  element-based – the loop variable takes on elements from the sequence that we're processing