Lecture 03
Iteration in Python

based in part on notes from the CS-for-All curriculum
developed at Harvey Mudd College
Last Time (lecture 02)

- Conditional Statements and Flow of Control
  - `if`
  - `if-else`
  - `if-elif-else`
  - True/False Blocks (require indentation)
- Variable Scope
  - Local
  - Global
- Memory: Frames and the Stack
  - Tracing global, local, and printed output
  - Functions calling Functions
Review
Default Index/Slicing Values

s == s[:]
s[:] = s[:]
s[: ] = s[0:len(s):1]
s[len(s):] = ‘ ’

s=’01234’ # len(s) == 5
s[2:] == s[ ?? : ?? : ?? ]
s[:3] == s[    :    :    ]
s[: :2] == s[    :    :    ]
s[:4:3] == s[    :    :    ]
s[1::2] == s[    :    :    ]
Review
Default Index/Slicing Values

\[
\begin{align*}
  s & == s[:]
  
  s[:] & == s[:]
  
  s[::] & == s[0:len(s):1]
  
  s[len(s):] & == ''
  
  s='01234'  \ # \ len(s) == 5
  
  s[2:] & == s[2:5:1]
  
  s[:3] & == s[0:3:1]
  
  s[::2] & == s[0:5:2]
  
  s[:4:3] & == s[0:4:3]
  
  s[1::2] & == s[1:5:2]
\end{align*}
\]
Lecture 03 Goals

- Introduce Test Driven Design (TDD)
- Iteration
  - Definite vs. Indefinite looping
- **for** loops
  - Element-based vs. Index-based
- List comprehensions
  - Generative vs. Manipulative
  - Uniform vs. Conditional
Test Driven Design

When coding

1. Think clearly about how each function should work
   • Inputs (what are arguments)
   • Outputs (what should be returned)
   • Special cases
   • Usual cases

2. Develop a function signature (def + docstring)

3. Write actual “test cases” before you start to code each function

4. Add/improve tests as needed

This approach is also known as Test First Design.
Write a function \( \text{gap}(x,y) \) that returns the distance between the numbers \( x \) and \( y \)? Use \textbf{if} statements and not a function like \texttt{abs} or \texttt{max}.

1. Think clearly about how each function should work

   • Inputs (what are arguments)
     Two numbers, \( x \) and \( y \)

   • Outputs (what should be returned)
     The distance between \( x \) and \( y \), i.e. \(|x-y|\)

   • Special cases
     If \( x==y \), must return 0

   • Usual cases
     \( x > y \) or \( x < y \)
Test Driven Design Example

Write a function \( \text{gap}(x,y) \) that returns the distance between the numbers \( x \) and \( y \)? Use \textbf{if} statements and not a function like \texttt{abs} or \texttt{max}.

\[ \]

2. Develop a function signature (def + docstring)

\[
\text{def } \text{gap}(x,y):
\begin{array}{c}
\text{'''Returns distance between two input numbers.'''}
\end{array}
\]

NOTE: The doc string should explain what the function does (and how to use it, i.e. inputs, outputs) but NOT how it does it.
Test Driven Design Example

Write a function gap(x,y) that returns the distance between the numbers x and y? Use if statements and not a function like abs or max.

3. Write actual “test cases” before you start code each function
   • Special cases: x==y must return 0
   • Usual case: x > y, x < y
   • Note the test cases go in a new function

```python
def gap_test():
    assert gap(10,10)==0, 'x==y test failed'
    assert gap(1, 10)==9, 'x<y test failed'
    assert gap(15,13)==2, 'x>y test failed'
```
Improving Tests

4. Add/improve tests as needed

- Creating student accounts for CS department machines
- The code was tested and it worked, but it failed to account for cases where there were two sections of the class on CAB (CS 4)
- **Edge case**- a case that will rarely happen, but your program should still be able to handle it
- For CS logins, add test to make sure it works for class with two sections
Test Driven Design

Now code/test your function, design will be informed by tests that need to pass.

def gap_test():
    assert gap(10,10)==0, 'x==y test failed'
    assert gap(1, 10)==9, 'x<y test failed'
    assert gap(15,13)==2, 'x>y test failed'

def gap(x, y): # Fill in after first set of tests!
    ''' Returns the distance between two input numbers.'''
    if x > y:
        return x - y
    else:
        return y - x

gap_test()

As you proceed keep testing,

4. Add/improve tests as needed
Write a function called `repeat_element(string, index, num_times)` that takes as input a string, the index of the element that we want to repeat, and the number of times we want to repeat. The function should return a new string in which the element of the string at position index is repeated `num_times` times.

1. Think clearly about how each function should work
   - Inputs (what are arguments)
   - Outputs (what should be returned)
   - Special cases
   - Usual cases

2. Develop a function signature (def + docstring)

3. Write actual “test cases” before you start to code each function.
Iteration: Loops

- A loop is a sequence of *instructions* to be repeated
- Definite and Indefinite
  - Definite: repeat exactly X times
  - Indefinite: repeat until some condition changes

This is Bijou. Bijou is demonstrating the following iteration examples:

```plaintext
for every front paw
    paw = paw + frilly blue glove
while sun == shining
    shed_more_fur()
```
Definite Loops

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* statements a specific number of times

- Example:

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

  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 in Python
for Loops (cont.)

• General syntax:

```python
for <variable> in <sequence>:
    <body of the loop>
```

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

• In this case, our sequence is a sequence of values, but it could be any sequence (i.e. `for word in list_of_words`)

• 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)
```

- **Flowchart Description**
  - **Decision Box:** Does the sequence have more values?
  - **Yes Path:**
    - Assign the next value in the sequence to the variable.
    - Execute the statements in the body.
  - **No Path:** Execute statement after the loop.
Executing Our Earlier Example (with one extra statement)

```python
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
```
Executing Our Earlier Example
(with one extra statement)

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

---

**Diagram:**

- **Flowchart Description:**
  - **Decision:** `does [1, 2, 3] have more values?`
    - **Yes:**
      - **Assign:** Assign the next value in the sequence to `i`
      - **Actions:**
        - `print('Warning')`
        - `print(i)`
    - **No:** `print('That's all.')`

**Output/Action:**

- `i`: 1
Executing Our Earlier Example
(with one extra statement)

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

<table>
<thead>
<tr>
<th>more?</th>
<th>i</th>
<th>output/action</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1</td>
<td>Warning 1</td>
</tr>
<tr>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

assign the next value in the sequence to `i`
Executing Our Earlier Example
(with one extra statement)

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

---

**Flowchart Description:**

- **Question:** Does [1, 2, 3] have more values?
- **Answer:** No
  - **Output:**
    - more? | i | output/action
    - yes | 1 | Warning
    - yes | 1 | That's all.

- **Answer:** Yes
  - **Action:**
    - Assign the next value in the sequence to `i`
    - `print('Warning')`
    - `print(i)`

---

21
Executing Our Earlier Example
(with one extra statement)

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

<table>
<thead>
<tr>
<th>more?</th>
<th>i</th>
<th>output/action</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1</td>
<td>Warning</td>
</tr>
<tr>
<td>yes</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Executing Our Earlier Example
(with one extra statement)

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

<table>
<thead>
<tr>
<th>more?</th>
<th>i</th>
<th>output/action</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>1</td>
<td>Warning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>yes</td>
<td>2</td>
<td>Warning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')

**skipping to end of loop**
for i in [1, 2, 3]:
    print('Warning')
    print(i)
print('That's all.')
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.
Simple Repetition Loops

• To repeat a loop's body $N$ times:

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

• What would this loop do?

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

• To repeat a loop's body $N$ times:

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

• Example:

```python
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:

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

• Example:

```python
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!
To repeat a loop's body $N$ times:

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

What would this loop do?

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

Output:

```
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:
  
  ```python
  for i in range(7):
      print(i * 5)
  
  How many repetitions?
  Output?
  ```
Simple Repetition Loops (cont.)

• Another example:

```python
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

• A definite loop is a loop in which the number of repetitions is fixed before the loop even begins.

• In a for loop, # of repetitions = len(sequence)

```python
for <variable> in <sequence>:
    <body of the loop>
```
To print the warning 20 times, how could you fill in the blank?

```python
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?

```python
for i in _________________:
    print('Warning!')
```

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

These are all sequences with a length of 20!
Python Arithmetic Shortcuts (language feature)

• Here are some *augmented assignment* statements that can be used in for loops!

• Consider this code:

  ```python
  age = 14
  age = age + 1
  ```

• Instead of writing

  ```python
  age = age + 1
  ```

  we can just write

  ```python
  age += 1
  ```
Python Arithmetic Shortcuts (cont.)

<table>
<thead>
<tr>
<th>shortcut</th>
<th>equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>var += expr</td>
<td>var = var + (expr)</td>
</tr>
<tr>
<td>var -= expr</td>
<td>var = var - (expr)</td>
</tr>
<tr>
<td>var *= expr</td>
<td>var = var * (expr)</td>
</tr>
<tr>
<td>var /= expr</td>
<td>var = var / (expr)</td>
</tr>
<tr>
<td>var //= expr</td>
<td>var = var //= (expr)</td>
</tr>
<tr>
<td>var %= expr</td>
<td>var = var % (expr)</td>
</tr>
<tr>
<td>var **= expr</td>
<td>var = var ** (expr)</td>
</tr>
</tbody>
</table>

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?

```python
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?

```python
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]))

Trace the execution of sum, determine the output

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  # returns 150

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

<table>
<thead>
<tr>
<th>x</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>40</td>
<td>100</td>
</tr>
<tr>
<td>50</td>
<td>150</td>
</tr>
</tbody>
</table>

no more values in vals, so we're done: **return: 150, output: 150**
Using a Loop to Sum a List of Numbers

```python
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]))
```

<table>
<thead>
<tr>
<th>x</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>40</td>
<td>100</td>
</tr>
<tr>
<td>50</td>
<td>150</td>
</tr>
</tbody>
</table>

no more values in `vals`, so we're done: **return: 150, output: 150**
Another Example

• What would this code output?

```python
num_iters = 0
for val in [2, 4, 16, 8, 10]:
    num_iters += 1
    print(val * 10)
print(num_iters)
```

• Use a table to help you:

<table>
<thead>
<tr>
<th>more?</th>
<th>val</th>
<th>num_iters</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another Example

• What would this code output?

```python
num_iters = 0
for val in [2, 4, 16, 8, 10]:
    num_iters += 1  # num_iters = num_iters + 1
    print(val * 10)
print(num_iters)
```

• Use a table to help you:

<table>
<thead>
<tr>
<th>more?</th>
<th>val</th>
<th>num_iters</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>yes</td>
<td>4</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>yes</td>
<td>16</td>
<td>3</td>
<td>160</td>
</tr>
<tr>
<td>yes</td>
<td>8</td>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>yes</td>
<td>10</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>no</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result
Index-Based for Loop

vals = [3, 15, 17, 7]

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

vals = [3, 15, 17, 7]
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
def sum(vals):
    # vals = [10, 20, 30, 40, 50]
    result = 0
    # initializer
    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)

<table>
<thead>
<tr>
<th>i</th>
<th>vals[i]</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>150</td>
</tr>
</tbody>
</table>

no more values in range(5), so we're done
return 150, output: 150
def sum(vals):
    result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result

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

What happens if we omit the initializer?

A. Nothing, it works fine
B. Undefined, initialized with random number
C. Error, local variable referenced before initialization
D. Python would look for global variable
def sum(vals):
    #result = 0
    for i in range(len(vals)):
        result += vals[i]
    return result

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

What happens if we omit the initializer?

A. Nothing, it works fine
B. Undefined, initialized with random number
C. Error, local variable referenced before initialization
D. Python would look for global variable
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]))

<table>
<thead>
<tr>
<th>i</th>
<th>vals[i]</th>
<th>vals[i - 1]</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

A. 0
B. 1
C. 2
D. 3
E. 7
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, 3, 3, 5]))    # print 3

<table>
<thead>
<tr>
<th></th>
<th>vals[i]</th>
<th>vals[i - 1]</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>B.</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>C.</td>
<td>2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>D.</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>E.</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
More on Cumulative Arithmetic

• Here's a loop-based factorial in Python:

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

• Does this function work?
More on Cumulative Arithmetic

• Here's a loop-based factorial in Python:

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

• Does this function work? No!

```python
def fac(n):
    result = 1
    for x in range(n) # [0,1,2,...,n-1]
        result *= x  # 1* 0 = 0...
    return result  # result = 0
More on Cumulative Arithmetic

• How can we make this do what we want?

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

Hint:

`range([start], stop[, step])`

- **start**: Starting number of the sequence.
- **stop**: Generate numbers up to, but not including this number.
- **step**: Difference between each number in the sequence.
More on Cumulative Arithmetic

• How can we make this do what we want?

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

Hint:

```python
range([start], stop[, step])
```

- **start**: Starting number of the sequence.
- **stop**: Generate numbers up to, but not including this number.
- **step**: Difference between each number in the sequence.
More on Cumulative Arithmetic

• Here's a loop-based factorial in Python:

```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
More on Cumulative Arithmetic

• Here's a loop-based factorial in Python:

```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?
Cumulative Arithmetic with Strings

• Let’s define an iterative `remove_vowels` function that takes in a string `s` and returns the string without any vowels:

```
def remove_vowels(s):
    # your code here!
```

• Examples:

```python
>>> s = remove_vowels('recurse')
>>> print(s)
'rcrs'
>>> s = remove_vowels('vowels')
>>> print(s)
'vwls'
```
Here's one loop-based version:

```python
def remove_vowels(s):
    result = ''  # the accumulator
    for c in s:
        if c not in 'aeiou':
            result += c  # accumulates the
    result
    print result
```
Cumulative Arithmetic with Strings (cont.)

• Here's one loop-based version:

```python
def remove_vowels(s):
    result = ''
    for c in s:
        if c not in 'aeiou':
            result += c
    return result
```

• Let's trace through `remove_vowels('vowels')`:

```python
s = 'vowels'
c result
```
Here's one loop-based version:

```python
def remove_vowels(s):
    result = ''
    for c in s:
        if c not in 'aeiou':
            result += c
    return result
```

Let's trace through `remove_vowels('vowels')`:

- `s = 'vowels'`
- `c`, `result`

<table>
<thead>
<tr>
<th>c</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'v'</td>
<td>'' + 'v' → 'v'</td>
</tr>
<tr>
<td>'o'</td>
<td>'v'    (no change)</td>
</tr>
<tr>
<td>'w'</td>
<td>'v' + 'w' → 'vw'</td>
</tr>
<tr>
<td>'e'</td>
<td>'vw'   (no change)</td>
</tr>
<tr>
<td>'l'</td>
<td>'vw' + 'l' → 'vwl'</td>
</tr>
<tr>
<td>'s'</td>
<td>'vwl' + 's' → 'vwls'</td>
</tr>
</tbody>
</table>
List Comprehensions

- List comprehensions use `for` loops within brackets to construct a list.
- We can create a list of integers up to i by using list comprehensions.

```
def create_list(size):
    result = [i for i in range(size)]
    return result

def squares(length):
    return [x**2 for x in range(length)]
```

- Format: `[expression for item in list]`
- The above syntax is useful for creating lists in one line. It includes all items in that list.
- You can also use list comprehensions to modify an existing list.

Why not just use ‘result = range(size)’?
List Comprehensions (cont.)

- We can include if-else statements to perform more complex operations.
- Let’s try the remove vowel function with list comprehensions.

```python
def remove_vowels(str):
    result = [c for c in str if c not in 'aeiou']
    return result
```

- This syntax allows us to use complex expressions to make a list in a single line.

- 2 valid formats:

  
  - [expression1 if condition else expression2 for item in list]
  - [expression for item in list if condition]
What is the output of the following expression?

def double_evens(int_list):
    return [2*i if i%2==0 else i for i in int_list]

double_evens([i for i in range(10)])

A. [0,1,4,3,8,5,12,7,16,9]
B. [0,1,2,3,4,5,6,7,8,9]
C. [0,1,4,3,4,5,12,7,16,9,20]
D. [0,4,8,12,16]
E. Error message
What is the output of the following expression?

def double_evens(int_list):
    return [2*i if i%2==0 else i for i in int_list]

double_evens([i for i in range(10)])

A. [0,1,4,3,8,5,12,7,16,9]
B. [0,1,2,3,4,5,6,7,8,9]
C. [0,1,4,3,4,5,12,7,16,9,20]
D. [0,4,8,12,16]
E. Error message
What does this program output?

```python
s = 'time to think! '
result = ''
for i in range(len(s)):
    if s[i - 1] == ' ':
        result += s[i]
print(result)
```

A. tt
B. ttt
C. tothink!
D. timetothink!
E. none of these
What does this program output?

```python
s = 'time to think!'
result = ''
for i in range(len(s)):
    if s[i - 1] == ' ':
        result += s[i]
print(result)
```

<table>
<thead>
<tr>
<th>i</th>
<th>s[i-1]</th>
<th>s[i]</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>''</td>
<td>'t'</td>
<td>'t'</td>
</tr>
<tr>
<td>1</td>
<td>'t'</td>
<td>'i'</td>
<td>'t'</td>
</tr>
<tr>
<td>2</td>
<td>'i'</td>
<td>'m'</td>
<td>'t'</td>
</tr>
<tr>
<td>3</td>
<td>'m'</td>
<td>'e'</td>
<td>'t'</td>
</tr>
<tr>
<td>4</td>
<td>'e'</td>
<td>''</td>
<td>'t'</td>
</tr>
<tr>
<td>5</td>
<td>''</td>
<td>'t'</td>
<td>'tt'</td>
</tr>
<tr>
<td>6</td>
<td>'t'</td>
<td>'o'</td>
<td>'tt'</td>
</tr>
<tr>
<td>7</td>
<td>'o'</td>
<td>''</td>
<td>'tt'</td>
</tr>
<tr>
<td>8</td>
<td>''</td>
<td>'t'</td>
<td>'tttt'</td>
</tr>
<tr>
<td>9</td>
<td>'t'</td>
<td>'h'</td>
<td>'tttt'</td>
</tr>
<tr>
<td>10</td>
<td>'h'</td>
<td>'i'</td>
<td>'tttt'</td>
</tr>
<tr>
<td>11</td>
<td>'i'</td>
<td>'n'</td>
<td>'tttt'</td>
</tr>
<tr>
<td>12</td>
<td>'n'</td>
<td>'k'</td>
<td>'tttt'</td>
</tr>
<tr>
<td>13</td>
<td>'k'</td>
<td>'!'</td>
<td>'tttt'</td>
</tr>
<tr>
<td>14</td>
<td>'!'</td>
<td>''</td>
<td>'tttt'</td>
</tr>
</tbody>
</table>

A. tt
B. ttt
C. tothink!
D. timetothink!
E. none of these
What does this program output?

```python
s = 'time to think! ' 
result = '' 
for i in range(len(s)):
    if s[i - 1] == ' ':
        result += s[i]
print(result)
```

Could you do the same thing using an element-based for loop?

```python
s = 'time to think! ' 
result = '' 
for c in s:
    if ________ == ' ':
        result += _____
print(result)
```
What does this program output?

```
s = 'time to think!'
result = ''
for i in range(len(s)):
    if s[i - 1] == ' ':
        result += s[i]
print(result)
```

Could you do the same thing using an `element-based` for loop?  `no`

```
s = 'time to think!'
result = ''
for c in s:
    if c == ' ':
        result += c
print(result)
```
vals = [3, 15, 17, 7]
def sum(vals):
    result = 0
    for x in vals:
        result += x
    return result

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

**element-based loop**

**index-based loop**
Meet UTA Alex Liu’s Nephew Wesley. When Wesley rests, we all rest
Side Note: Staying on the Same Line When Printing

• By default, print puts an invisible newline character at the end of whatever it prints.
  • causes separate prints to print on different lines

• Example: What does this output?

```python
for i in range(7):
    print(i * 5)
```
Side Note: Staying on the Same Line When Printing

• By default, `print` puts an invisible `newline` character at the end of whatever it prints.
  • causes separate `print`s to print on different lines

• Example: What does this output?

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

  0
  5
  10
  15
  20
  25
  30
Staying on the Same Line When Printing (cont.)

• To get separate prints to print on the same line, we can replace the newline with something else.

• Examples:

```python
    for i in range(7):
        print(i * 5, end=' ')  
    0 5 10 15 20 25 30 

    for i in range(7):
        print(i * 5, end=',' ) 
    0,5,10,15,20,25,30, 
```