Computational Thinking

Lecture 05: Loops

University of Engineering and Technology
VIETNAM NATIONAL UNIVERSITY HANOI

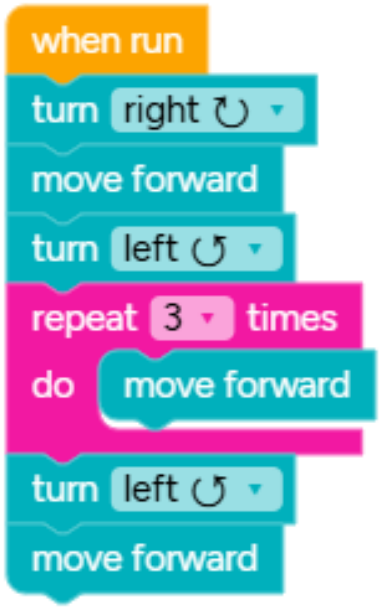
Outline

- For Loops
- The Accumulator Pattern
- Ranges
- Looping over Indices (vs. Items)
- Nested Lists & Nested For Loops
- Operations on Nested Lists
- Tabular Data as Nested Lists

For Loops

Remember this game?

- Get the bird to the pig and avoid the TNT




```
...
for i in range(3):
    move_forward()
...
```

How to Print Each Item in List?

print each item of the list lst

```
print("The list contains:")
print(lst[0]) # print item at index 0
print(lst[1]) # print item at index 1
 # ...but when does it end ????
```

We Need Repetition

- So far our programs just follow instructions **the same number of times** no matter what
  - Each statement in function executes exactly once per function call
  - Even if we skip some statements with if-elif-else statements, we're still executing the if-elif-else exactly once.
  - Even if we call a function multiple times, we're still calling the function a fixed number of times
- **But programs need to repeat actions!**
  - Print every item in a list of unknown size
  - Send financial statement to all customers
  - Check spelling errors for all words in document
  - Move all the tokens on a board in a game
- We need a way to **control** repetition based on inputs…

For Loops

Syntax

```
for <var> in <seq>:
    <statement>
    …
    <statement>
```

Example

```
for item in lst:
    print(item)
```

Execution (simplified):

- Assign the *1st* element of <seq> to <var>. Execute all the indented statements.
- Assign the *2nd* element of <seq> to <var>. Execute all the indented statements again.
- …
- Assign the *last* element of <seq> to <var>. Execute all the indented statements one last time.

Terminology

Syntax

```
for <var> in <seq>:
    <statement>
    …
    <statement>
```

Example

```
for item in lst:
    print(item)
```

loop variable

loop sequence

loop body

How to Print Each Item in List?

```
print("The list contains:")
# print each item of lst on own line
for item in lst:
    print(item)
```

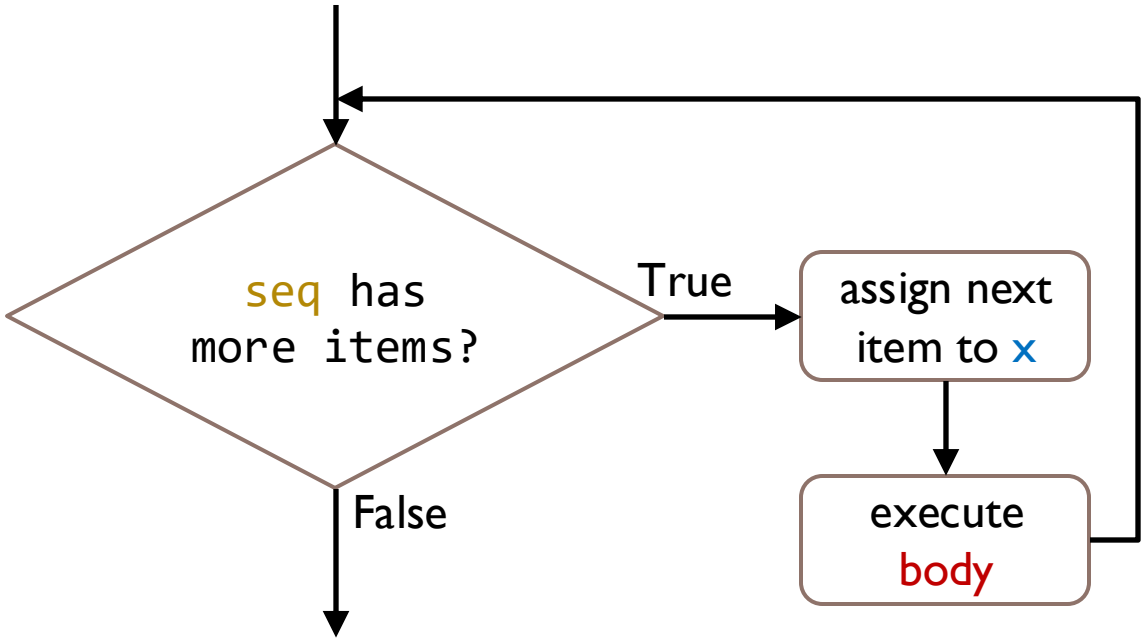
Execution (simplified):

- Assign the *1st* element of lst to item. Execute all the print statement.
- Assign the *2nd* element of lst to item. Execute all the print statement again.
- …
- Assign the *last* element of lst to item. Execute all the print statement one last time.

Execution (No Longer Simplified)

```
for x in seq:
    body
```

This diagram is another **flowchart**



The check for "more items?" happens n+1 times if there are n items.

Strings Are Also Sequences

```
print("The string contains:")
# print each char of s on own line
for char in s:
    print(char)
```

Execution (simplified):

- Assign the *first* element of s to char. Execute all the print statement.
- Assign the *second* element of s to char. Execute all the print statement again.
- …
- Assign the *last* element of s to char. Execute all the print statement one last time.

Question

How many times will line 3 be executed when this code segment is executed?

```
1 # Precondition: s is a string
2 for v in 'aeiou':
3     if v in s:
4         return True
5     return False
```

A: Exactly 5 times
B: 1 to 5 times
C: Exactly `len(s)` times
D: 1 to `len(s)` times
E: Never

The Accumulator Pattern

Compute the Average of Numeric List

Express algorithm as pseudocode

```
def avg(lst):
    """Returns: the average of all elements in lst.
    Precondition: lst is a non-empty list and each item
    is a float or int."""

    # 1. Set num to the sum of all the elements of lst

    # 2. Set den to the number of elements of lst

    # 3. Return num/den

    return 0
```

Compute the Average of Numeric List

Elaborating pseudocode into code: working **iteratively***

```
def avg(lst):
    """Returns: the average of all elements in lst.
    Precondition: lst is a non-empty list and each item
    is a float or int."""

    # 1. Set num to the sum of all the elements of lst
    num = 0 # TODO
    # 2. Set den to the number of elements of lst
    den = len(lst)
    # 3. Return num/den
    return 0
```

* Implement what I know how to do. When I don't know how, create a variable and assign it something that is partially right.

Compute the Average of Numeric List

Elaborating pseudocode into code: focusing on sum

```
...
# 1. Set num to the sum of all the elements of lst
num = 0
for x in lst:
    num = num + x
...
```

Execution:

- Assign the *first* element of lst to x. Add it into num.
- Assign the *second* element of lst to x. Add it into num.
- …
- Assign the *last* element of lst to x. Add it into num.

Compute the Average of Numeric List

Finished code

```
def avg(lst):
    """Returns: the average of all elements in lst.
    Precondition: lst is a non-empty list and each item
    is a float or int."""

    # 1. Set num to the sum of all the elements of lst
    num = 0
    for x in lst:
        num = num + x
    # 2. Set den to the number of elements of lst
    den = len(lst)
    # 3. Return num/den
    return num/den
```

Compute the Average of Numeric List

Accumulator

```
def avg(lst):
    """Returns: the average of all elements in lst.
    Precondition: lst is a non-empty list and each item
    is a float or int."""

    # 1. Set num to the sum of all the elements of lst
    num = 0
    for x in lst:
        num = num + x
    # 2. Set den to the number of elements of lst
    den = len(lst)
    # 3. Return num/den
    return num/den
```

num is an **accumulator** variable

Accumulator Pattern

An **accumulator** is a variable a loop uses to help compute a value, one item at a time.

```
accumulator = <initial value>
for item in seq:
    ...
    accumulator = accumulator <operator> item ...
    ...
# accumulator now contains the desired value
```

```
num = 0
for x in lst:
    num = num + x
# num now contains the sum
```

More Accumulator Examples

See accumulators.py in demo code:

- Compute the sum of a list
- Compute the product of a list
- Remove the spaces from a string

Ranges

How to Print 1 through n?

Pseudocode

```
def count(n):
    """Prints 1, 2, ..., n; each number on its own line.
    Precondition: n is an int >= 1."""

    # Create the list 1..n
    # Loop through the list and print each item

    pass
```

How to Print 1 through n?

Elaborating pseudocode into code: working **iteratively***

```
def count(n):
    """Prints 1, 2, ..., n; each number on its own line.
    Precondition: n is an int >= 1."""

    # Create the list [1, 2, ..., n]
    nums = [1] # TODO

    # Loop through nums and print each number
    for num in nums:
        print(num)
```

* Implement what I know how to do. When I don't know how, create a variable and assign it something that is partially right.

How to Print 1 through n?

Elaborating pseudocode into code: focusing on nums

```
# Create the list [1, 2, ..., n]
nums = [1] # TODO
```

Question

Why does this attempt not work correctly?

```
# Create the list [1, 2, ..., n]
nums =  []
for i in n:
    nums.append(i)
```

A. Because we should have converted n to a string.
B. Because integers are not sequences.
C. Because a method, like append(), cannot take a loop variable as an argument.
D. Because nums is an accumulator but we did not assign to it in the loop body.
E. Actually, it does work correctly.

Ranges

A **range** is a sequence of numbers.
Because it is a **sequence**, it is an object that supports len() and indexing/slicing.

```
>>> r = range(3)
>>> r
range(0, 3)
>>> list(r)
[0, 1, 2]
>>> list(range(1, 3))
[1, 2]
>>> len(r)
3
```

```
>>> r.start
0
>>> r.stop
3
>>> r[0]
0
>>> r[1]
1
```

https://docs.python.org/3/library/stdtypes.html

How to Print 1 through n?

Elaborating pseudocode into code: focusing on nums

```
# Create the list [1, 2, ..., n]
nums = list(range(1, n+1))
```

How to Print 1 through n?

Finished code

```
def count(n):
    """Prints 1, 2, ..., n; each number on its own line.
    Precondition: n is an int >= 1."""

    # Create the list [1, 2, ..., n]
    nums = list(range(1, n+1))

    # Loop through nums and print each number
    for num in nums:
        print(num)
```

How to Print 1 through n?

Simplified code (option 1):
no need for separate nums variable

```
def count(n):
    """Prints 1, 2, ..., n; each number on its own line.
    Precondition: n is an int >= 1."""

    # Create the sequence 1..n and print each number
    for num in range(1, n+1):
        print(num)
```

How to Print 1 through n?

Simplified code (option 2): use helper function to print

```
def count(n):
    """Prints 1, 2, ..., n; each number on its own line.
    Precondition: n is an int >= 1."""

    # Create the list [1, 2, ..., n]
    nums = list(range(1, n+1))

    # Print the list
    print_list(nums)
```

How to Print 1 through n?

Simplified code (option 3): combine both previous options

```
def count(n):
    """Prints 1, 2, ..., n; each number on its own line.
    Precondition: n is an int >= 1."""

    print_list(list(range(1, n+1)))
```

But reducing our code to the minimum number of lines is not a goal and might be counterproductive.
Go with what you think will be most readable for other beginning programmers now, not with what is shortest!

Looping over Indices (vs. Items)

How to Increment Every List Element?

Let's try looping over the items

```
def incr_list_wrong(nums):
    """Adds 1 to every element of nums.
    Example: changes input list [6,0,7] to [7,1,8].
    Precondition: nums list contains only numbers."""

    #  Loop through list and increment each number
    # WARNING: incorrect code
    for n in nums:
        n = n + 1
```

```
>>> lst = [6, 0, 7]
>>> incr_list_wrong(lst)
>>> lst
[6, 0, 7]
```

Review: For Loops

Syntax

```
for <var> in <seq>:
    <statement>
    …
    <statement>
```

Example

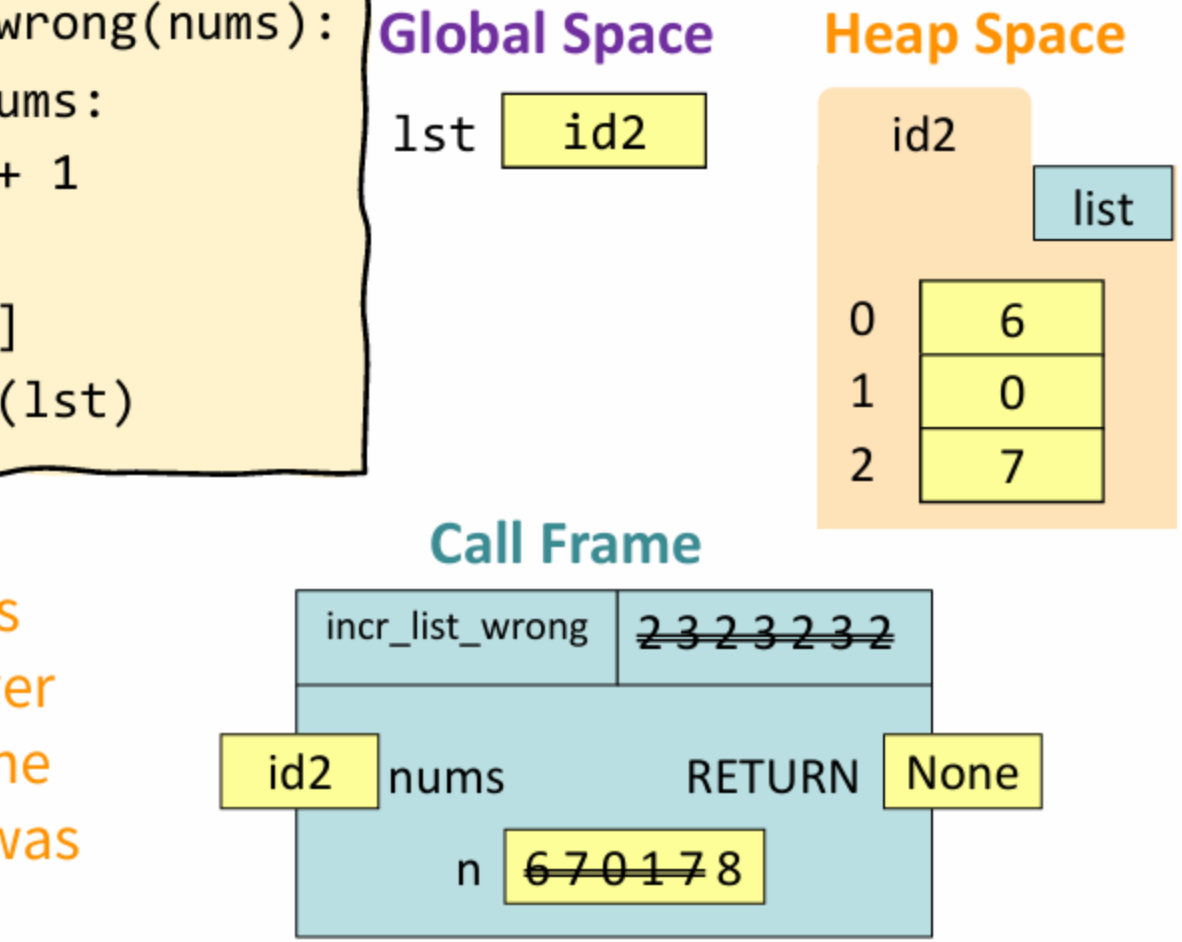
```
for item in lst:
    print(item)
```

Execution (simplified):

- Assign the *1st* element of <seq> to <var>. Execute all the indented statements.
- Assign the *2nd* element of <seq> to <var>. Execute all the indented statements again.
- …
- Assign the *last* element of <seq> to <var>. Execute all the indented statements one last time.

Execution Diagram — Just Before Return

```
1 def incr_list_wrong(nums):
2     for n in nums:
3         n = n + 1
4
5 lst = [6, 0, 7]
6 incr_lst_wrong(lst)
```



At no time was the list object ever changed! Only the loop variable n was changed.

Review: Ranges

A **range** is a sequence of numbers.
Because it is a **sequence**, it is an object that supports len() and indexing/slicing.

```
>>> r = range(3)
>>> r
range(0, 3)
>>> list(r)
[0, 1, 2]
```

```
>>> r[0]
0
>>> r[1]
1
```

https://docs.python.org/3/library/stdtypes.html

How to Increment Every List Element?

Let's try looping over the indices

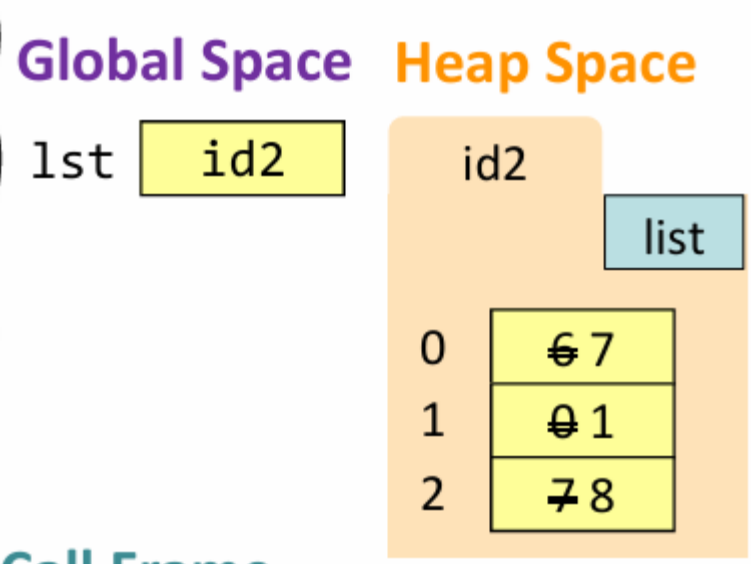
```
def incr_list (nums):
    """Adds 1 to every element of nums.
    Example: changes input list [6,0,7] to [7,1,8].
    Precondition: nums list contains only numbers."""

    for i in range(len(nums)):
        nums[i] = nums[i] + 1
```

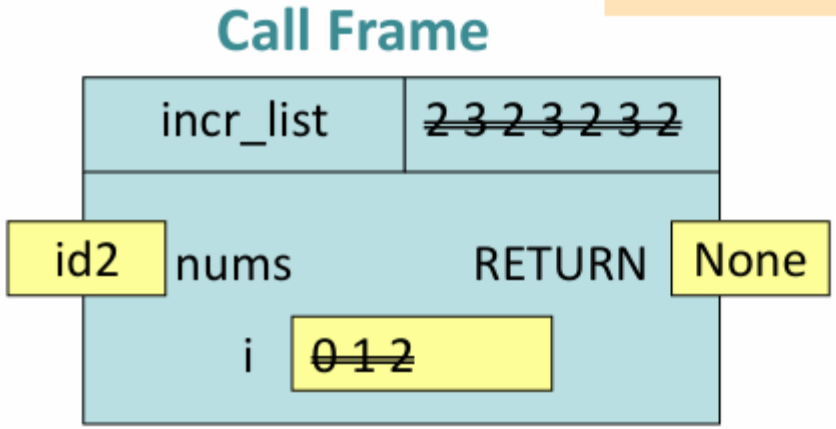
```
>>> lst = [6, 0, 7]
>>> incr_list(lst)
>>> lst
[7, 1, 8]
```

Execution Diagram — Just Before Return

```
1 def incr_list(nums):
2     for i in range(len(nums)):
3         nums[i] = nums[i] + 1
4
5 lst = [6, 0, 7]
6 incr_lst(lst)
```



The loop variable is changed, **and** the list object is changed!



Two Patterns for List Looping

Loop over Items

```
for <var> in <list>:
    # use <var>
```

Example

```
lst = [1, 2, 3]
for item in lst:
    print(item)
```

- Simple code
- Less powerful
- Do it when you **can**

Loop over Indices

```
for <var> in range(len(<list>)):
    # use <list>[<var>]
```

Example

```
lst = [1, 2, 3]
for idx in range(len(lst)):
    print(lst[idx])
```

- More complicated code
- More powerful
- Do it when you **must**

Enumerate

enumerate() function adds a counter to each item in a list or any other iterable, and returns a list of tuples containing the index position and the element for each element of the iterable.

main.py

```
a = ["Geeks", "for", "Geeks"]

# Iterating list using enumerate
# to get both index and element
for i, name in enumerate(a):
    print(f"Index {i}: {name}")

# Converting to a list of tuples
print(list(enumerate(a)))
```

Run | Share | $ | Command Line Arguments

```
Index 0: Geeks
Index 1: for
Index 2: Geeks
[(0, 'Geeks'), (1, 'for'), (2, 'Geeks')]

** Process exited - Return Code: 0 **
```

https://www.geeksforgeeks.org/python/enumerate-in-python/

Nested Lists & Nested For Loops

Sometimes Just a List is Not Enough

Data	Organization
Outline of an essay	Bullets, sub-bullets, sub-sub-bullets
Contents of a book	Chapter, paragraph, sentence, word
Video	Frames, pixels
Spreadsheet	Rows and columns

These are all **hierarchical**:
lists of lists, lists of lists of lists, …

Nested Lists

List elements can be any object.

And lists are objects.

Therefore, lists can contain other lists!
(And those lists can themselves contain lists, and so on.)

Example:

```
[[2, 4],
 [3, 6, 9],
 [4, 8, 12, 16]]
```

One outer list with three elements

Three **inner** lists,
each with a different number of elements

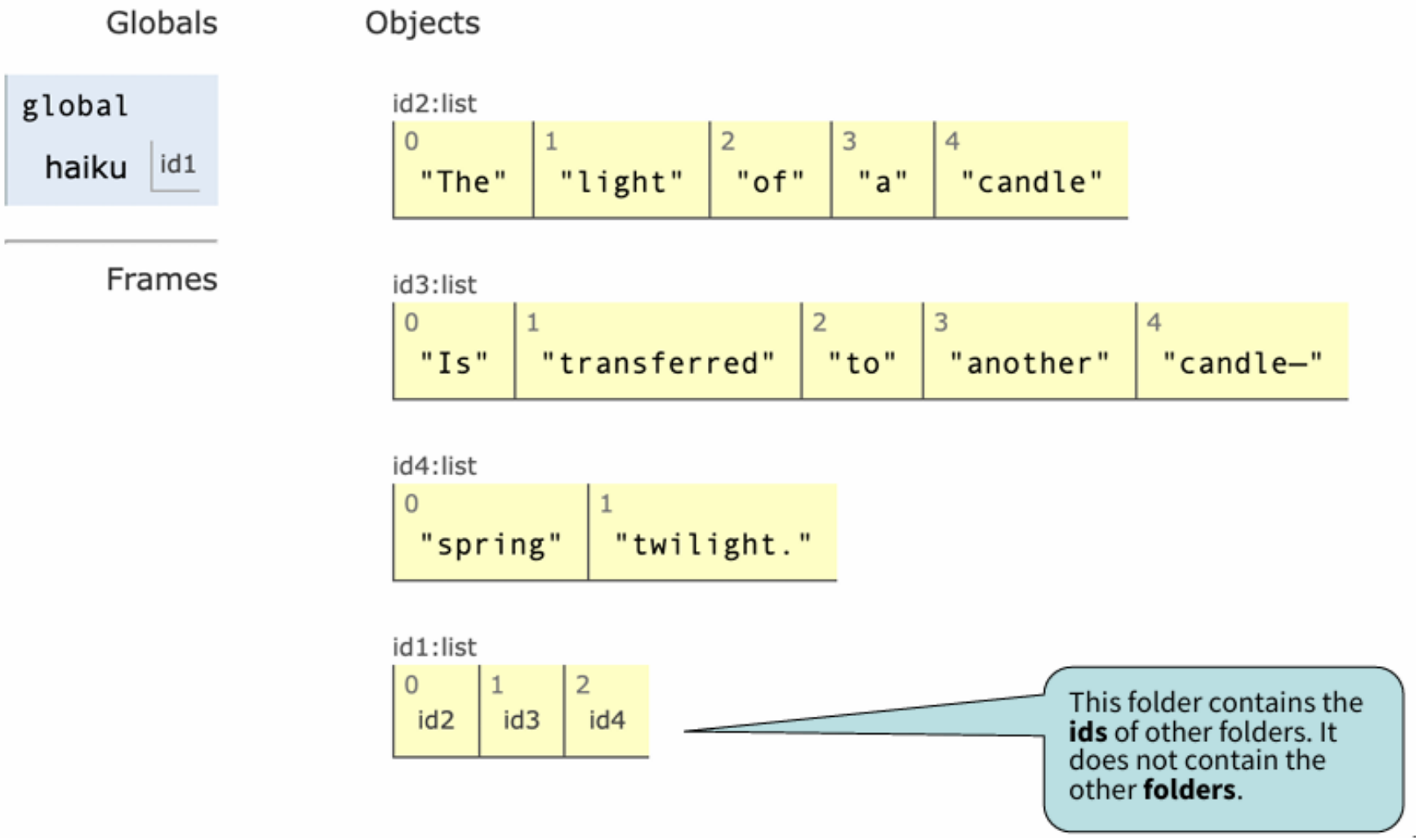
A Haiku by Yosa Buson

The light of a candle
Is transferred to another candle—
spring twilight.

A spellchecker might treat it as a nested list:

```
[['The', 'light', 'of', 'a', 'candle'],
 ['Is', 'transferred', 'to', 'another', 'candle—'],
 ['spring', 'twilight.']]
```

Memory structure



How to Print Each Word of Haiku?

With a helper function from previous page

```
def print_list(lst):
    """Print lst, a list of strings."""
    for word in lst:
        print(word)

def print_haiku_with_helper(h):
    """Print haiku h, a list of list of strings."""
    for line in h:
        print_list(line)
```

How to Print Each Word of Haiku?

With **nested for loops**

```
def print_haiku(h):
    """Print haiku h, a list of list of strings."""
    for line in h:
        for word in line:
            print(word)
```

loop	Loop variable	Loop sequence
Outer	line	h
Inner	word	line

Operations on Nested Lists

Nested List Length and Indexing

```
>>> h = [['The', 'light', 'of', 'a', 'candle'], ['Is',
'transferred', 'to', 'another', 'candle–'], ['spring',
'twilight.']]
>>> len(h)
```

Length of outer list

```
3
>>> len(h[0])
```

Length of an inner list

```
5
>>> h[2][0]
```

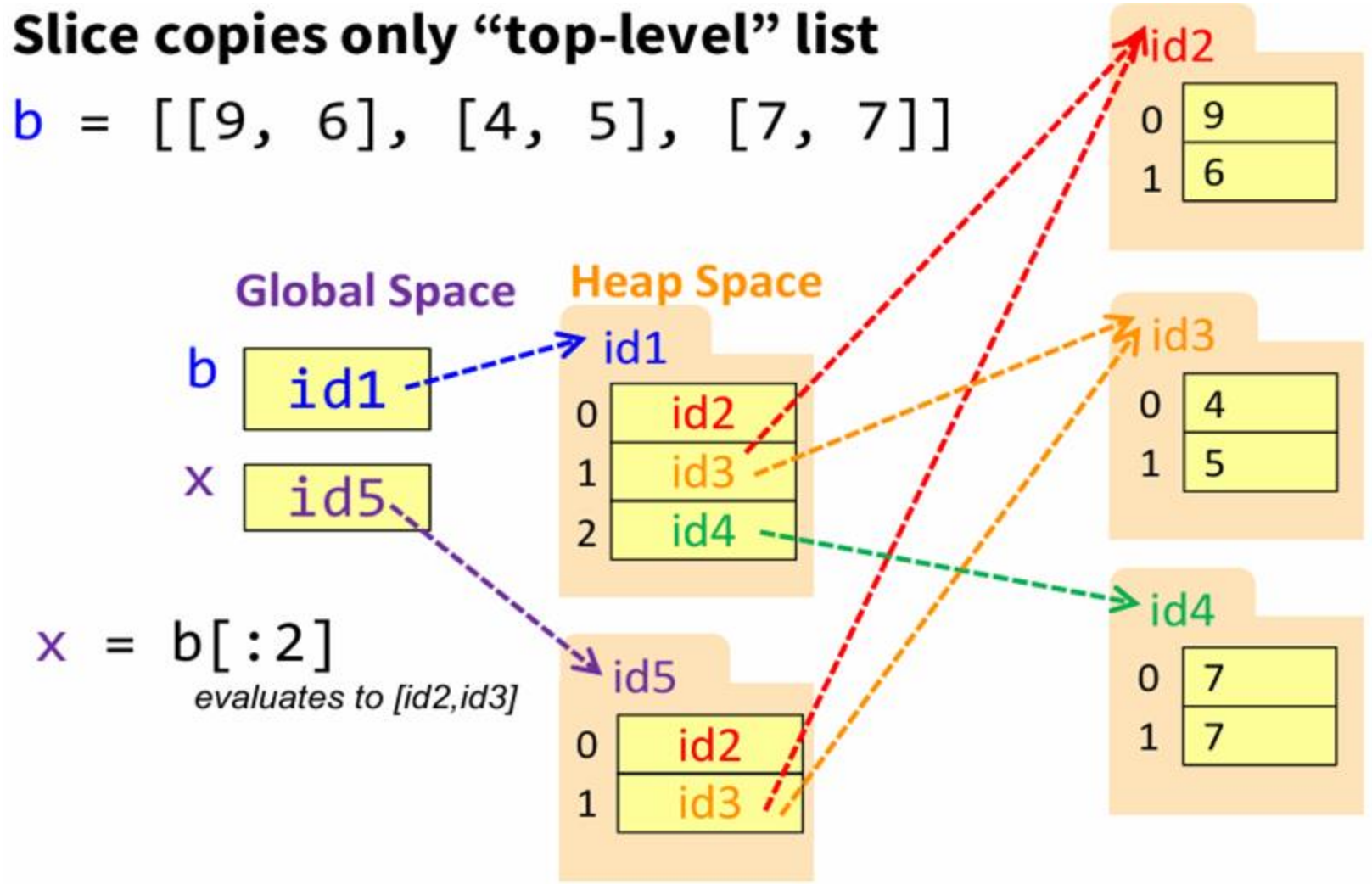
Access element

```
'spring'
>>> h[2][1] = 'dawning'
```

Assign to element

```
>>> h
[['The', 'light', 'of', 'a', 'candle'], ['Is', 'transferred',
'to', 'another', 'candle–'], ['spring', dawning.']]
```

Nested List Slicing



Question

What is the output of the code segment?

```
>>> b = [[9,6],[4,5],[7,7]]
>>> x = b[:2]
>>> x[1].append(10)
>>> x
```

A. [[9,6,10]]
B. [[9,6],[4,5,10]]
C. [[9,6],[4,5,10],[7,7]]
D. [[9,6],[4,10],[7,7]]

Tabular Data as Nested Lists

Tabular Data — Spreadsheet

How could you represent a table using nested lists?

Column major:

every column is an inner list

NetID	A1	A2	A3
abc1	100	95	97
def2	100	80	92

```
[['NetID','abc1','def2'],
 ['A1', 100, 100],
 ['A2', 95, 80],
 ['A3', 97, 92]]
```

Row major:

every row is an inner list

NetID	A1	A2	A3
abc1	100	95	97
def2	100	80	92

```
[['NetID','A1','A2','A3'],
 ['abc1', 100, 95, 97],
 ['def2', 100, 80, 92]]
```

Both representations are feasible and common.

Rectangular vs. Ragged 2D Lists

2D list: a list of lists

Rectangular: every inner list has same length

Ragged: not rectangular

Ragged:

```
[['The', 'light', 'of', 'a', 'candle'],
 ['Is', 'transferred', 'to', 'another', 'candle–'],
 ['spring', 'twilight.']]
```

Rectangular:

```
[['NetID','A1','A2','A3'],
 ['abc1', 100, 95, 97],
 ['def2', 100, 80, 92]]
```

How to Average a Table of Numbers?

Using nested for loops, the accumulator pattern, and looping over items

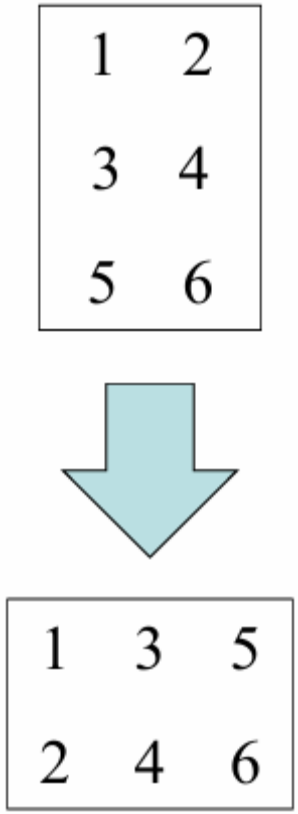
```
def avg_tab(tab):
    """Returns the average of tab, a 2D list of numbers."""
    sum = 0
    count = 0
    for inner_list in tab:
        for number in inner_list:
            sum = sum + number
            count = count + 1
    return sum / count
```

How to Add 1 to Every Number in Table?

Using nested for loops, looping over indices, and nested indexing

```
def add1_tab (tab):
    """Adds 1 to every number in tab,
       a 2D list of numbers."""
    for outer_idx in range(len(tab)):
        for inner_idx in range(len(tab[outer_idx])):
            old_val = tab[outer_idx][inner_idx]
            tab[outer_idx][inner_idx] = old_val + 1
```

How to Convert Between Row/Col Major?



Aka matrix **transpose**

Summary - Key Takeaways

- **For loops & ranges:** Iterate items with `for x in seq`; use `range(start, step)` for counted iterations.
- **Accumulator pattern:** Initialize once, `e.g., total=0, out=[]`, update inside the loop to build results.
- **Items vs. indices:** Prefer iterating **items**; use **indices** only for in-place mutation or when positions matter
- **Nested lists & loops:** Model tables as lists-of-lists; traverse `for row in table: for cell in row`.
- **Tabular data:** Aggregate, filter, and map without unintended mutation; handle ragged rows and aliasing carefully.