

Lecture 8: Containers & Sequences

CS 61A - Summer 2024
Raymond Tan

Sequences

A **sequence** is an ordered collection of values.

Examples:

```
"hello world"  
"abcdefghijkl"
```

Strings

Sequence of
characters

```
[1, 2, 3, 4, 5]  
[True, "hi", 0]
```

Lists

Sequence of values of
any data type

Lists

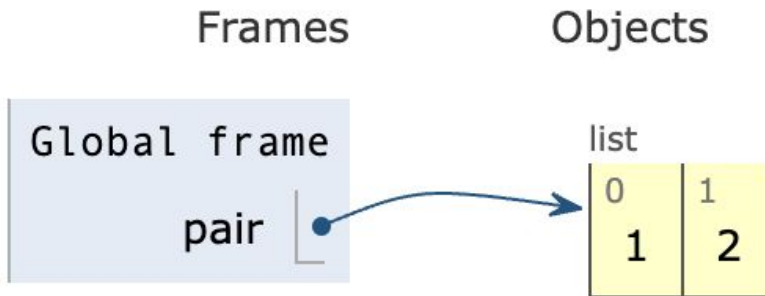
Lists

- A list is an **ordered sequence** of elements
- Some operations we can perform on lists:
 - Access an element at a certain position
 - Get the number of elements in a list
 - Concatenate two lists together into one
 - Determine if an element exists inside the list
 - ... and much more
- Lists can contain more than one datatype as elements
 - Including other lists!

Demo: Working with Lists

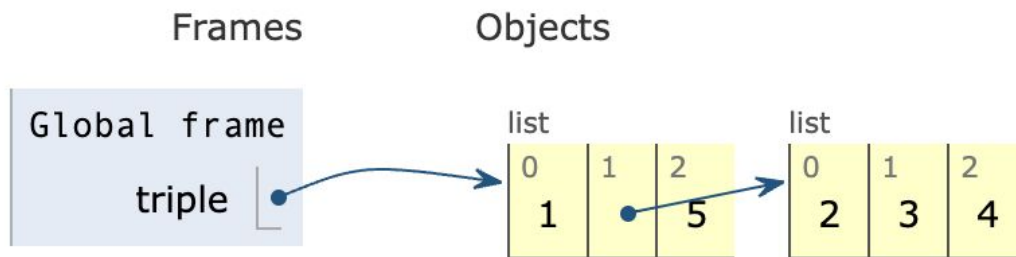
Box and Pointer Notation

- Environment diagrams allow us to visualize the contents of a list
- Each box contains either a primitive value, or points to a compound value
 - Ex: `pair = [1, 2]`



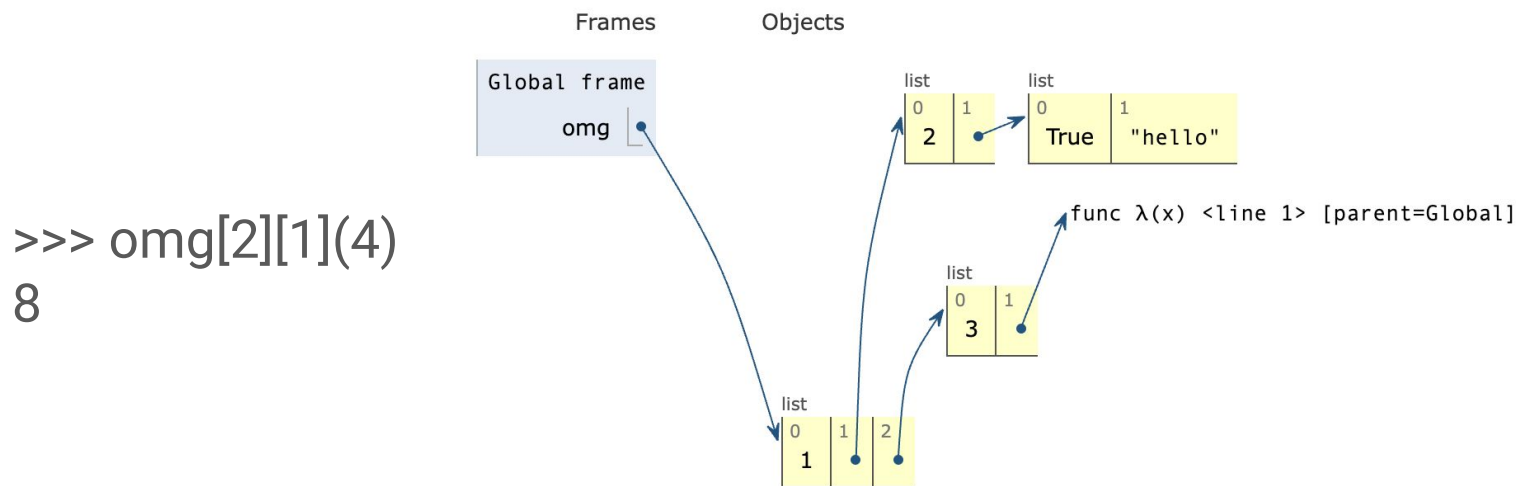
Box and Pointer Notation

- Environment diagrams allow us to visualize the contents of a list
- Each box contains either a primitive value, or points to a compound value
 - Ex: `triple = [1, [2, 3, 4], 5]`



Box and Pointer Notation

- Environment diagrams allow us to visualize the contents of a list
- Each box contains either a primitive value, or points to a compound value
 - Ex: `omg = [1, [2, [True, 'hello']], [3, lambda x : 2 * x]]`



List Slicing

- List slicing returns a specified “chunk” of a list
- Syntax:

```
lst[i:j:k]
```

i: Starting Index
(**inclusive**)

j: Ending Index
(**exclusive**)

k (optional):
Step size
(default set to 1)

Demo: List Slicing

For Statements

Example: count

- Problem: We want to come up with a function that takes two arguments - `s` and `value`. The goal is to return the number of times the integer `value` appears in the list `s`.

count: while loop

```
def count(s, value):  
    """Return the number of times that  
    value appears in the list s."""  
    total, index = 0, 0  
    while index < len(s):  
        element = s[index]  
        if element == value:  
            total += 1  
        index += 1  
    return total
```

Is there a way we can
make this implementation
shorter?

count: for statement

```
def count(s, value):  
    """Return the number of times that  
    value appears in the list s."""  
    total = 0  
    for element in s:  
        if element == value:  
            total += 1  
    return total
```

The for statement automatically:

- Binds the current element in the list to a variable
- Removes the need to keep track of an index, as the variable bound to the current element will automatically rebound to the next element

For Statements

- A `for` statement is a way of iterating over sequences
- General syntax:

```
for <var> in <iterable>:  
    # Body of the loop
```

- `var` is bound to the current value in the iterable
- `iterable` is the object we're iterating over (ex: list, but we'll talk about many more iterables later)

Range Objects

- A range object returns a sequence of values created by calling the range function
- Default values are set for a range depending on how many arguments we pass in
 - If one argument is passed in, this represents the end value (exclusive), starting from 0
 - If two arguments are passed in, this represents the start value (inclusive) and end value (exclusive)
 - If three arguments are passed in, this represents the start value (inclusive), the end value (exclusive), and the step size

Demo: Range Objects

For statements using range objects

- In many different problems we'll encounter, we'll need not only the element of a list, but also the *index* in which that element is stored
 - Ex: Printing out the indices of a list that store a given value, x
- It's common to iterate over a range object where the argument to range is the length of the list
 - This allows you to iterate over a set of all indices belonging to that list
 - Syntax: `for i in range(len(s))`

List comprehensions

- List comprehensions allow us to initialize a **list** based on another **iterable** in a single line
- Syntax:

```
[<expression> for <element> in <sequence>]  
[<expression> for <element> in <sequence> if <conditional>]
```

List comprehensions

```
[<expression> for <element> in <sequence>]  
[<expression> for <element> in <sequence> if <conditional>]
```

- expression - The expression we want to include in the final list
- element - The variable bound to where we currently are in the sequence
- sequence - The iterable we are basing the list comprehension on
- conditional (optional) - Only include expression if this conditional is true

Break

Example: Index evens

- Write a function that takes in a list `s`, and returns a list of all the indices of the elements in the list for which the element is an even number.
- Try doing it with a traditional for loop, and then with a list comprehension!

index evens - Traditional For Loop

```
def index_evens(s):  
    ans = []  
    for i in range(len(s)):  
        if s[i] % 2 == 0:  
            ans = ans + [i]  
    return ans
```

index evens - List Comprehension

```
def index_evens(s):  
    return [i for i in range(len(s)) if s[i] % 2 == 0]
```


Strings

Strings are an Abstraction

Representing Data:

`'False', 'Everyday', '2500'`

Representing Language:

`'Entschuldigung , wie bitte' , 'Nos vemos' , '你好'`

Representing Programs:

`'curry = lambda f: lambda x: lambda y: f(x,y)'`

Three Forms of Strings

Single quotes strings and double quotes strings are equivalent

`'Hello there' , "General Kenobi"`

Multi-line strings automatically insert new lines

`"""Shall I compare thee to a summer's day?`

`Thou art more lovely and more temperate"""`

The `\n` is an escape sequence signifying a line feed

`"""Shall I compare thee to a summer's day?\nThou art more lovely and more temperate"""`

Multi-line strings are often used in docstrings as they usually span multiple lines

Strings are Sequences

- A String can be thought of as a sequence of characters
 - We can get the number of elements in the sequence by using `len`
 - We can index into a String to get an individual character
 - Note: An element of a String is itself a String, just a single element
 - This is different from a list, where if we have a list of numbers, indexing into that list would return a number
- However, the `in` and `not in` operators can match substrings, rather than only individual elements in the sequence

```
>>> 'a' in 'snapple'
True
>>> 'app' in 'snapple'
True
```

Strings are Sequences

- Can use a `for` statement to iterate over the characters of a String

```
word = 'cs61a'  
for c in word:  
    print(c)
```

This would print out
the individual
characters of
'cs61a'

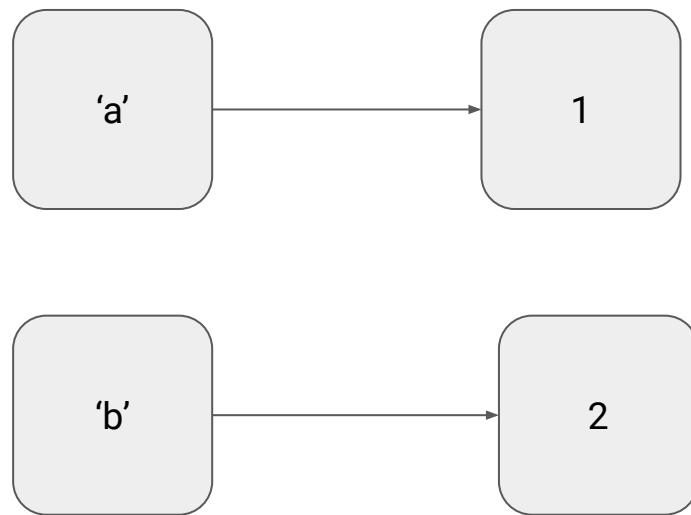
Dictionaries

Dictionaries

- Dictionaries are an example of a key-value data structure
 - Each data entry consists of a (key, value) pair
- Incredibly efficient to retrieve data as it utilizes a concept known as **hashing** (out of scope)

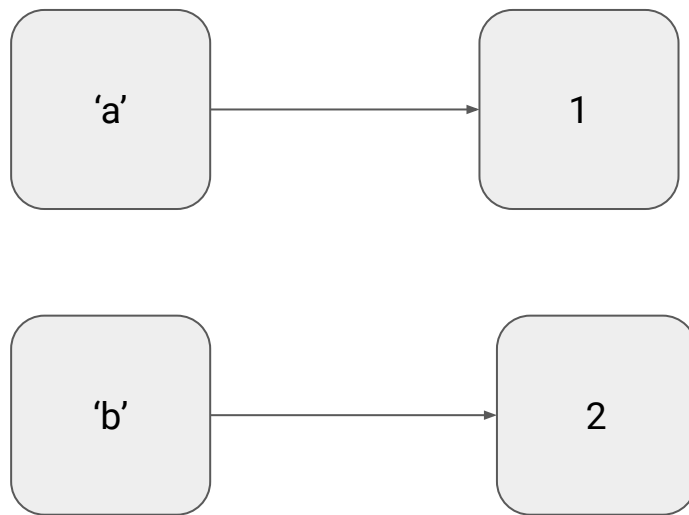
Dictionaries - Example

```
[>>> d = {}  
[>>> d['a'] = 1  
[>>> d['b'] = 2
```



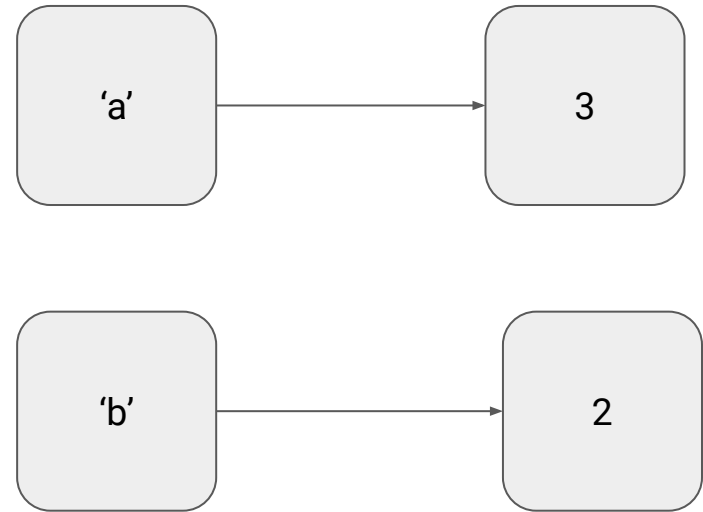
Dictionaries - Example

```
[>>> d['a']  
1  
>>> d['b']  
2  
>>> d['c']  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'c'
```



Dictionaries - Example

```
[>>> d[ 'a ' ] = 3  
[>>> d[ 'a ' ]  
3
```



Dictionaries - Example

```
[>>> len(d)
```

```
2
```

```
[>>> del d['b']
```

```
[>>> len(d)
```

```
1
```



Summary

- A sequence is an ordered collection of values
 - Ex: Lists, Strings, ranges, dictionaries
- Box and Pointer Notation is how we represent lists in diagrams
 - Primitive values are stored in the boxes directly, while compound values are represented with an arrow pointer
- For statements are a more direct way of iteration over an iterable
- List comprehensions allow us to create lists in a single line based on another iterable
- Dictionaries are an efficient, key-value store datatype