Containers
Announcements
Box-and-Pointer Notation
The Closure Property of Data Types
The Closure Property of Data Types

• A method for combining data values satisfies the closure property if:

  The result of combination can itself be combined using the same method.
The Closure Property of Data Types

- A method for combining data values satisfies the closure property if:
  
  The result of combination can itself be combined using the same method

- Closure is powerful because it permits us to create hierarchical structures
The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:
  - The result of combination can itself be combined using the same method
- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on
The Closure Property of Data Types

• A method for combining data values satisfies the closure property if:
  The result of combination can itself be combined using the same method

• Closure is powerful because it permits us to create hierarchical structures

• Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else)
Box-and-Pointer Notation in Environment Diagrams
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element.

Each box either contains a primitive value or points to a compound value.
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element.
Each box either contains a primitive value or points to a compound value.

```
pair = [1, 2]
```
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element. Each box either contains a primitive value or points to a compound value.

\[
\text{Global frame} \\
\text{pair} \\
\text{list} \\
\begin{array}{c}
0 \\
1 \\
2 \\
\end{array}
\]

pair = [1, 2]
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element. Each box either contains a primitive value or points to a compound value.

```
1  pair = [1, 2]
2
3  nested_list = [[1, 2], [],
4      [[3, False, None],
5      [4, lambda: 5]]]
```
Slicing

(Demo)
Slicing Creates New Values

```python
digits = [1, 8, 2, 8]
start = digits[:1]
middle = digits[1:3]
end = digits[2:]
full = digits[:]
```

Diagram showing how slicing creates new values in a list.
Processing Container Values
Aggregation

Several built-in functions take iterable arguments and aggregate them into a value.
Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- `sum(iterable[, start])` -> value

  Return the sum of an iterable (not of strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.
Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum(iterable[, start]) -> value**
  
  Return the sum of an iterable (not of strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

- **max(iterable[, key=func]) -> value**
  
  With a single iterable argument, return its largest item.
  With two or more arguments, return the largest argument.
Several built-in functions take iterable arguments and aggregate them into a value

- `sum(iterable[, start]) -> value`

  Return the sum of an iterable (not of strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

- `max(iterable[, key=func]) -> value`
  `max(a, b, c, ..., key=func) -> value`

  With a single iterable argument, return its largest item.
  With two or more arguments, return the largest argument.

- `all(iterable) -> bool`

  Return True if bool(x) is True for all values x in the iterable.
  If the iterable is empty, return True.
Strings
Strings are an Abstraction
Strings are an Abstraction

Representing data:

'200'    '1.2e-5'    'False'    '[1, 2]'
Strings are an Abstraction

Representing data:

'200'    '1.2e-5'    'False'    '[1, 2]'  

Representing language:

""""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.""""
Strings are an Abstraction

Representing data:

'200'   '1.2e-5'   'False'   ' [1, 2]'

Representing language:

"""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name."""

Representing programs:

'curry = lambda f: lambda x: lambda y: f(x, y)'
Strings are an Abstraction

Representing data:

'200'    '1.2e-5'    'False'    '[1, 2]'  

Representing language:

""""""And, as imagination bodies forth
The forms of things unknown, the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name."""

Representing programs:

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

(Demo)
String Literals Have Three Forms

```python
>>> 'I am string!
'I am string!

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'
```
String Literals Have Three Forms

>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'

Single-quoted and double-quoted strings are equivalent
String Literals Have Three Forms

>>> 'I am string!
'I am string!

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'

>>> """"The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python
claims, Readability counts.
Read more: import this.'
String Literals Have Three Forms

>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'

>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'

A backslash "escapes" the following character

Single-quoted and double-quoted strings are equivalent
String Literals Have Three Forms

>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'

>>> """The Zen of Python claims, Readability counts. Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'

A backslash "escapes" the following character

"Line feed" character represents a new line
Dictionaries

{"Dem": 0}
Limitations on Dictionaries
Limitations on Dictionaries

Dictionaries are collections of key-value pairs
Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:
Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary cannot be a list or a dictionary (or any mutable type)
Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary **cannot be** a list or a dictionary (or any mutable type)

• Two keys **cannot be equal**; There can be at most one value for a given key
Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)

• Two **keys cannot be equal**; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries
Limitations on Dictionaries

Dictionaries are collections of key–value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary **cannot be** a list or a dictionary (or any mutable type)

• Two **keys cannot be equal**; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction
Limitations on Dictionaries

Dictionaries are collections of key-value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary **cannot be** a list or a dictionary (or any mutable type)

• Two **keys cannot be equal**; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value
Dictionary Comprehensions
Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```
Dictionary Comprehensions

{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

Short version: {<key exp>: <value exp> for <name> in <iter exp>}
Dictionary Comprehensions

\[
\{\text{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}\}
\]

Short version: \[
\{\text{<key exp>: <value exp> for <name> in <iter exp>}\}
\]

An expression that evaluates to a dictionary using this evaluation procedure:
Dictionary Comprehensions

\{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>\}

Short version: \{<key exp>: <value exp> for <name> in <iter exp>\}

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
Dictionary Comprehensions

\{<key \text{ exp}>: <value \text{ exp}> \text{ for } <name> \text{ in } <\text{iter \text{ exp}}> \text{ if } <\text{filter \text{ exp}}>\}\n
Short version: \{<key \text{ exp}>: <value \text{ exp}> \text{ for } <name> \text{ in } <\text{iter \text{ exp}}>\}\n
An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty result dictionary that is the value of the expression
Dictionary Comprehensions

\{<\text{key exp}> : <\text{value exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}> \text{ if } <\text{filter exp}>\}\}

Short version: \{<\text{key exp}> : <\text{value exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}>\}\}

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty \text{result dictionary} that is the value of the expression
3. For each element in the iterable value of \text{iter exp}:
Dictionary Comprehensions

\{<key \ exp>: <value \ exp> \ for \ <name> \ in \ <iter \ exp> \ if \ <filter \ exp>\}\n
Short version: \{<key \ exp>: <value \ exp> \ for \ <name> \ in \ <iter \ exp>\}\n
An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty result dictionary that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1
Dictionary Comprehensions

{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}

Short version: {<key exp>: <value exp> for <name> in <iter exp>}

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty result dictionary that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1
   B. If <filter exp> evaluates to a true value, then add to the result dictionary an entry that pairs the value of <key exp> to the value of <value exp>
Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

Short version: `{<key exp>: <value exp> for <name> in <iter exp>}`

An expression that evaluates to a dictionary using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty *result dictionary* that is the value of the expression
3. For each element in the iterable value of `<iter exp>`:
   A. Bind `<name>` to that element in the new frame from step 1
   B. If `<filter exp>` evaluates to a true value, then add to the result dictionary an entry that pairs the value of `<key exp>` to the value of `<value exp>`

```
{x * x: x for x in [1, 2, 3, 4, 5] if x > 2}  evaluates to  {9: 3, 16: 4, 25: 5}
```
Example: Indexing

Implement `index`, which takes a sequence of `keys`, a sequence of `values`, and a two-argument `match` function. It returns a dictionary from `keys` to lists in which the list for a key `k` contains all `values` `v` for which `match(k, v)` is a true value.

```python
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which match(k, v) is a true value.
    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
    return __________________________________________________________
```
Example: Indexing

Implement `index`, which takes a sequence of `keys`, a sequence of `values`, and a two-argument `match` function. It returns a dictionary from `keys` to lists in which the list for a key `k` contains all `values v` for which `match(k, v)` is a true value.

```python
def index(keys, values, match):
    """Return a dictionary from keys k to a list of values v for which match(k, v) is a true value.
    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
    return {k: list(filter(lambda v: match(k, v), values)) for k in keys}
```
Example: Indexing

Implement `index`, which takes a sequence of `keys`, a sequence of `values`, and a two-argument `match` function. It returns a dictionary from `keys` to lists in which the list for a key `k` contains all `values` `v` for which `match(k, v)` is a true value.

```python
def index(keys, values, match):
    """Return a dictionary from keys `k` to a list of values `v` for which `match(k, v)` is a true value.
    >>> index([7, 9, 11], range(30, 50), lambda k, v: v % k == 0)
    {7: [35, 42, 49], 9: [36, 45], 11: [33, 44]}
    """
    return {k: [v for v in values if match(k, v)] for k in keys}
```