Containers
Announcements
Lists

['Demo']
Working with Lists
Working with Lists

>>> digits = [1, 8, 2, 8]
Working with Lists

```python
>>> digits = [1, 8, 2, 8]
>>> digits = [2//2, 2+2+2, 2, 2*2*2]
```
Working with Lists

```python
>>> digits = [1, 8, 2, 8]  
>>> digits = [2//2, 2+2+2, 2, 2*2*2]
```

The number of elements
Working with Lists

```python
>>> digits = [1, 8, 2, 8]
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]
```

**The number of elements**

```python
>>> len(digits)
4
```
Working with Lists

```python
>>> digits = [1, 8, 2, 8]
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]

The number of elements

>>> len(digits)
4

An element selected by its index
Working with Lists

```python
>>> digits = [1, 8, 2, 8]

The number of elements
>>> len(digits)
4

An element selected by its index
```
Working with Lists

>>> digits = [1, 8, 2, 8]

The number of elements

```python
>>> len(digits)
4
```

An element selected by its index

```python
>>> digits[3]
8
```

```python
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]
```

```python
>>> digits[3]
8
```
Working with Lists

```python
>>> digits = [1, 8, 2, 8]

The number of elements
```  
```python
>>> len(digits)
4
```  
An element selected by its index
```python
>>> digits[3]
8
```  
Concatenation and repetition
```python
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]
```  
```python
>>> getitem(digits, 3)
8
```
Working with Lists

```python
>>> digits = [1, 8, 2, 8]

The number of elements
```

```python
>>> len(digits)
4
```

```python
An element selected by its index
```

```python
>>> digits[3]
8
```

```python
Concatenation and repetition
```

```python
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```
Working with Lists

```python
>>> digits = [1, 8, 2, 8]

The number of elements
```n
```python
>>> len(digits)
4
```

An element selected by its index
```python
>>> digits[3]
8
```

Concatenation and repetition
```python
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

```python
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]
```

```python
>>> getitem(digits, 3)
8
```

```python
>>> add([2, 7], mul(digits, 2))
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```
Working with Lists

```python
>>> digits = [1, 8, 2, 8]

The number of elements

```python
>>> len(digits)
4

An element selected by its index

```python
>>> digits[3]
8

Concatenation and repetition

```python
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

Nested lists

```python
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]

>>> getitem(digits, 3)
8

>>> add([2, 7], mul(digits, 2))
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```
Working with Lists

```python
>>> digits = [1, 8, 2, 8]

The number of elements
>>> len(digits)
4

An element selected by its index
>>> digits[3]
8

Concatenation and repetition
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

Nested lists
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30

>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]

>>> getitem(digits, 3)
8

>>> add([2, 7], mul(digits, 2))
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]```
Containers
Containers
Containers

Built-in operators for testing whether an element appears in a compound value
Containers

Built-in operators for testing whether an element appears in a compound value

```python
>>> digits = [1, 8, 2, 8]
```
Containers

Built-in operators for testing whether an element appears in a compound value

```python
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
```
Containers

Built-in operators for testing whether an element appears in a compound value

```python
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
```
Containers

Built-in operators for testing whether an element appears in a compound value

```python
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
```
Containers

Built-in operators for testing whether an element appears in a compound value

```python
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
>>> not(5 in digits)
True
```
Containers

Built-in operators for testing whether an element appears in a compound value

```python
>>> digits = [1, 8, 2, 8]
>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
>>> not(5 in digits)
True
```

(Demo)
For Statements

(Demo)
Sequence Iteration
def count(s, value):
    total = 0
    for element in s:
        if element == value:
            total = total + 1
    return total
Sequence Iteration

def count(s, value):
    total = 0
    for element in s:
        if element == value:
            total = total + 1
    return total

Name bound in the first frame of the current environment (not a new frame)
For Statement Execution Procedure
For Statement Execution Procedure

for <name> in <expression>:
    <suite>
For Statement Execution Procedure

```python
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a sequence)
For Statement Execution Procedure

```python
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a sequence)

2. For each element in that sequence, in order:
For Statement Execution Procedure

for <name> in <expression>:
    <suite>

1. Evaluate the header <expression>, which must yield an iterable value (a sequence)

2. For each element in that sequence, in order:

   A. Bind <name> to that element in the current frame
For Statement Execution Procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a sequence)

2. For each element in that sequence, in order:
   
   A. Bind `<name>` to that element in the current frame

   B. Execute the `<suite>`
Sequence Unpacking in For Statements
Sequence Unpacking in For Statements

```python
classes = [[1, 2], [2, 2], [3, 2], [4, 4]]
same_count = 0
```
Sequence Unpacking in For Statements

```
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
```
Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```python
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0

>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1

>>> same_count
2
```
Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```python
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]

>>> same_count = 0

>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1

>>> same_count
2
```
Sequence Unpacking in For Statements

```python
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]

>>> same_count = 0

>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1

>>> same_count
2
```
Ranges
The Range Type

A range is a sequence of consecutive integers.*
The Range Type

A range is a sequence of consecutive integers.*

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers. *

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

..., −5, −4, −3, −2, −1, 0, 1, 2, 3, 4, 5, ...

range(−2, 2)

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]

\`range(-2, 2)`

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

...,-5,-4,-3,-2,-1,0,1,2,3,4,5,...

ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]

\[
\text{range}(-2, 2)
\]

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

...,-5,-4,-3,-2,-1,0,1,2,3,4,5,...

* Ranges can actually represent more general integer sequences.

Length: ending value – starting value
The Range Type

A range is a sequence of consecutive integers.*

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

Length: ending value - starting value

Element selection: starting value + index

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

Length: ending value – starting value

Element selection: starting value + index

```python
generate list(range(-2, 2))
[-2, -1, 0, 1]
generate list(range(4))
[0, 1, 2, 3]
```

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

... ..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

Length: ending value – starting value

Element selection: starting value + index

```python
>>> list(range(-2, 2))
[-2, -1, 0, 1]
```

```python
>>> list(range(4))
[0, 1, 2, 3]
```

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

... , -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

Length: ending value - starting value

Element selection: starting value + index

>>> list(range(-2, 2))
[-2, -1, 0, 1]

List constructor

>>> list(range(4))
[0, 1, 2, 3]

Range with a 0 starting value

*Ranges can actually represent more general integer sequences.*
The Range Type

A range is a sequence of consecutive integers.*

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]

Length: ending value - starting value

Element selection: starting value + index

\[
>>> \text{list(range(-2, 2))}
\]

[-2, -1, 0, 1]

List constructor

\[
>>> \text{list(range(4))}
\]

[0, 1, 2, 3]

Range with a 0 starting value

* Ranges can actually represent more general integer sequences.
List Comprehensions
List Comprehensions

```python
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
>>> [letters[i] for i in [3, 4, 6, 8]]
```
List Comprehensions

```python
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
>>> [letters[i] for i in [3, 4, 6, 8]]
['d', 'e', 'm', 'o']
```
List Comprehensions
List Comprehensions

\[
[\text{map exp} \ for \ <name> \ in \ <iter exp> \ if \ <filter exp>]
\]
List Comprehensions

\[
[\text{map exp} \ for \ <name> \ in \ <iter \ exp> \ if \ <filter \ exp>]
\]

Short version: \[\text{map exp} \ for \ <name> \ in \ <iter \ exp>\]
List Comprehensions

\[
\left[ \text{<map exp> for <name> in <iter exp> if <filter exp>} \right]
\]

Short version: \[
\left[ \text{<map exp> for <name> in <iter exp>} \right]
\]

A combined expression that evaluates to a list using this evaluation procedure:
List Comprehensions

\[
\left[ \text{map exp} \ for \ <\text{name}> \ in \ <\text{iter exp}> \ if \ <\text{filter exp}> \right]
\]

Short version: \[
\left[ \text{map exp} \ for \ <\text{name}> \ in \ <\text{iter exp}> \right]
\]

A combined expression that evaluates to a list using this evaluation procedure:

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

\[
[\text{map exp} \ for \ <name> \ in \ <iter \ exp> \ if \ <filter \ exp>]
\]

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

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
2. Create an empty result list that is the value of the expression
List Comprehensions

\[
[\text{map exp} \ for \ <name> \ in \ <iter \ exp> \ if \ <filter \ exp>]
\]

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

A combined expression that evaluates to a list using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty result list that is the value of the expression
3. For each element in the iterable value of \(<iter \ exp>\):
List Comprehensions

\[
\begin{align*}
&[\text{map exp} \ for \ <name> \ in \ <iter \ exp> \ if \ <filter \ exp>] \\
&\text{Short version: } [\text{map exp} \ for \ <name> \ in \ <iter \ exp>]
\end{align*}
\]

A combined expression that evaluates to a list using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty result list 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
List Comprehensions

\[
[\text{\langle map exp\rangle for \langle name\rangle in \langle iter exp\rangle if \langle filter exp\rangle}]
\]

Short version: \[\text{\langle map exp\rangle for \langle name\rangle in \langle iter exp\rangle}\]

A combined expression that evaluates to a list using this evaluation procedure:

1. Add a new frame with the current frame as its parent
2. Create an empty result list that is the value of the expression
3. For each element in the iterable value of \langle iter exp\rangle:
   
   A. Bind \langle name\rangle to that element in the new frame from step 1
   
   B. If \langle filter exp\rangle evaluates to a true value, then add the value of \langle map exp\rangle to the result list
Example: Promoted
Implement **promoted**, which takes a sequence `s` and a one-argument function `f`. It returns a list with the same elements as `s`, but with all elements `e` for which `f(e)` is a true value ordered first. Among those placed first and those placed after, the order stays the same.

```python
def promoted(s, f):
    """Return a list with the same elements as s, but with all elements e for which f(e) is a true value placed first."

    >>> promoted(range(10), odd)  # odds in front
    [1, 3, 5, 7, 9, 0, 2, 4, 6, 8]
    """
    return [e for e in s if f(e)] + [e for e in s if not f(e)]
```

```bash
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
[ 1, 3, 5, 7, 9, 0, 2, 4, 6, 8 ]
```