Containers
Announcements
Lists

['Demo']
Working with Lists
Working with lists

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

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

>>> 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
```
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]
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
```

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

```python
>>> digits
```

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

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

```python
>>> getitem(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]
```
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]
```  
```python
>>> digits = [2//2, 2+2+2+2, 2, 2*2*2]
```
```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
>>> len(digits)
4

An element selected by its index
>>> 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
```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
>>> 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
```

```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

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

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

\begin{code}
  for <name> in <expression>:
    <suite>
\end{code}

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
>>> 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
```
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]]
```

```python
>>> same_count = 0
```

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

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

A sequence of fixed-length sequences

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

```python
>>> same_count = 0
```

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

```python
>>> 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.*

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

* 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.*

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

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

A range is a sequence of consecutive integers.*

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

Ranges can actually represent more general integer sequences.*

**Length:** ending value – starting value

* 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

* 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.*

... \(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots\)

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(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.*

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

Length: ending value - starting value

Element selection: starting value + index

* Ranges can actually represent more general integer sequences.

List constructor

Range with a 0 starting value

Demo
Recursive Sums
Sum (recursively)

def mysum(L):
    if (L == []):
        return 0
    else:
        return L[0] + mysum(L[1:])

mysum([2, 4, 1, 5])

2 + mysum([4, 1, 5])
4 + mysum([1, 5])
1 + mysum([5])
5 + mysum([])
0
# --- DRILL ---
# Write an iterative function that takes as input
# integer “n” and returns the sum of the first “n”
# integers: sum(5) returns 1+2+3+4+5
# --- DRILL ---
# Write an iterative function that takes as input
# integer “n” and returns the sum of the first “n”
# integers: sum(5) returns 1+2+3+4+5

def sum_iter(n):
    sum = 0
    for i in range(0,n+1):
        sum = sum + i

    return( sum )
# --- DRILL ---
# Write a recursive function that takes as input
# integer “n” and returns the sum of the first “n”
# integers: sum(5) returns 1+2+3+4+5
# --- DRILL ---
# Write a recursive function that takes as input
# integer “n” and returns the sum of the first “n”
# integers: sum(5) returns 1+2+3+4+5

def sum_rec(n):
    if n == 0:
        return 0
    else:
        return n + sum_rec(n-1)
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]]
```

```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

[<map exp> for <name> in <iter exp> if <filter exp>]
List Comprehensions

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

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

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

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

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

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

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

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

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: \[<map \ exp> \ for \ <name> \ in \ <iter \ exp>\]

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

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<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>:
   A. Bind <name> to that element in the new frame from step 1
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 \textit{result list} that is the value of the expression
3. For each element in the iterable value of \texttt{<iter exp>}:
   
   A. Bind \texttt{<name>} to that element in the new frame from step 1
   
   B. If \texttt{<filter exp>} evaluates to a true value, then add the value of \texttt{<map exp>}
      to the result list
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

>>> '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

```python
>>> '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.'
```

Single-quoted and double-quoted strings are equivalent
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'

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

Single-quoted and double-quoted strings are equivalent.

A backslash "escapes" the following character.
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'

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

- Single-quoted and double-quoted strings are equivalent
- A backslash "escapes" the following character
- "Line feed" character represents a new line
Reversing a String
Reversing a List (recursively)

reverse("ward") = "draw"

reverse("ward") = reverse("ard") + "w"
    reverse("ard") = reverse("rd") + "a"
    reverse("rd") = reverse("d") + "r"
    reverse("d") = "d"
Reversing a List (recursively)

reverse("ward") = "draw"

reverse("ward") = reverse("ard") + "w"

reverse("ard") = "d" + "r" + "a"
Reversing a List (recursively)

```python
def reverse(s):
    if len(s) == 1:
        return s
    else:
        return reverse(s[1:]) + s[0]
```