61A Lecture 12

Monday, September 30

Announcements

Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm

Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm
- Optional Hog Contest due Thursday 10/3 @ 11:59pm

Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm
- Optional Hog Contest due Thursday 10/3 @ 11:59pm
- Homework 4 due Tuesday 10/8 @ 11:59pm

Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm
- Optional Hog Contest due Thursday 10/3 @ 11:59pm
- Homework 4 due Tuesday 10/8 @ 11:59pm
- Project 2 due Thursday 10/10 @ 11:59pm

Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm
- Optional Hog Contest due Thursday 10/3 @ 11:59pm
- Homework 4 due Tuesday 10/8 @ 11:59pm
- Project 2 due Thursday 10/10 @ 11:59pm
- Guerrilla Section 2 this Saturday 10/5 & Sunday 10/6 10am–1pm in Soda

Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm
- Optional Hog Contest due Thursday 10/3 @ 11:59pm
- Homework 4 due Tuesday 10/8 @ 11:59pm
- Project 2 due Thursday 10/10 @ 11:59pm
- Guerrilla Section 2 this Saturday 10/5 & Sunday 10/6 10am–1pm in Soda
  - Topics: Data abstraction, sequences, non–local assignment

Announcements

- Homework 3 due Tuesday 10/1 @ 11:59pm
- Optional Hog Contest due Thursday 10/3 @ 11:59pm
- Homework 4 due Tuesday 10/8 @ 11:59pm
- Project 2 due Thursday 10/10 @ 11:59pm
- Guerrilla Section 2 this Saturday 10/5 & Sunday 10/6 10am–1pm in Soda
    - Topics: Data abstraction, sequences, non-local assignment
    - Meet outside Soda 306

For Statements

(Demo)

Sequence Iteration

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:
```

Name bound in the first frame of the current environment (not a new frame)

```
        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

```
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 first frame of the current environment.

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 first frame of the current environment.
   B. Execute the <suite>.

Sequence Unpacking in For Statements

Sequence Unpacking in For Statements

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))

>>> same_count = 0
```

Sequence Unpacking in For Statements

A sequence of
fixed-length sequences

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))

>>> same_count = 0
```

Sequence Unpacking in For Statements

A sequence of
fixed-length sequences

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (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

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))

>>> same_count = 0
```

A name for each element in a fixed-length sequence

```
>>> 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

```
>>> pairs = ((1, 2), (2, 2), (2, 3), (4, 4))

>>> same_count = 0
```

A name for each element in a fixed-length sequence

Each name is bound to a value, as in multiple assignment

```
>>> 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.*

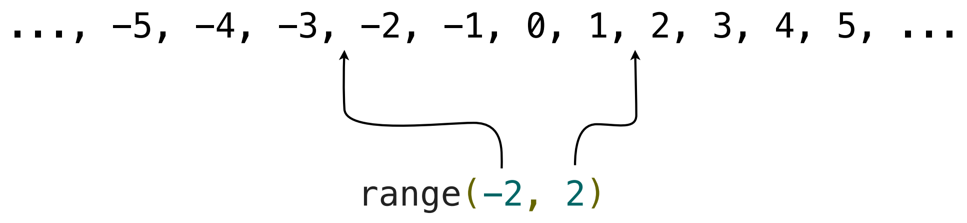
```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
```

```
range(-2, 2)
```

* Ranges can actually represent more general integer sequences.

The Range Type

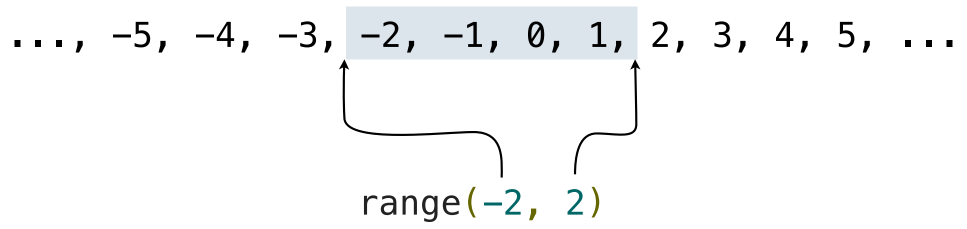
A range is a sequence of consecutive integers.*



* Ranges can actually represent more general integer sequences.

The Range Type

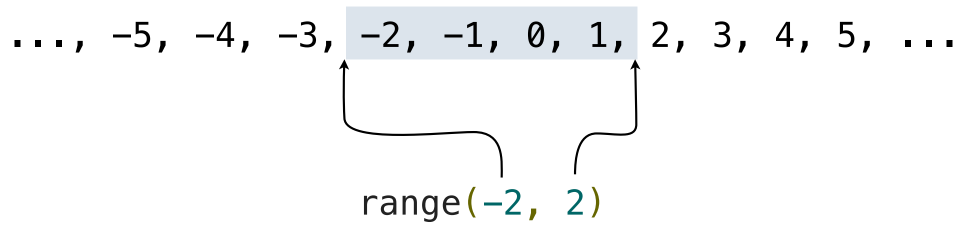
A range is a sequence of consecutive integers.*



* 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, ...
```

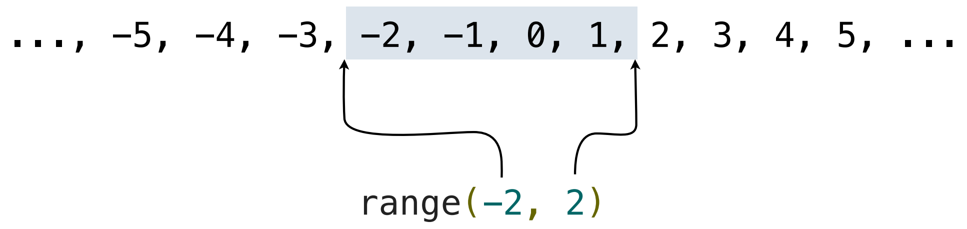
```
range(-2, 2)
```

Length: ending value - starting value

* Ranges can actually represent more general integer sequences.

The Range Type

A range is a sequence of consecutive integers.*



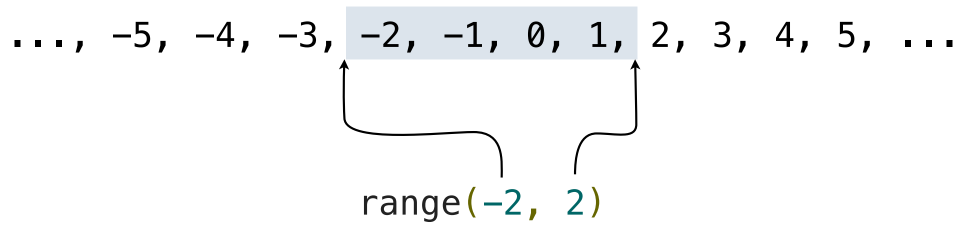
Length: ending value - starting value

Element selection: starting value + index

* Ranges can actually represent more general integer sequences.

The Range Type

A range is a sequence of consecutive integers.*



Length: ending value - starting value

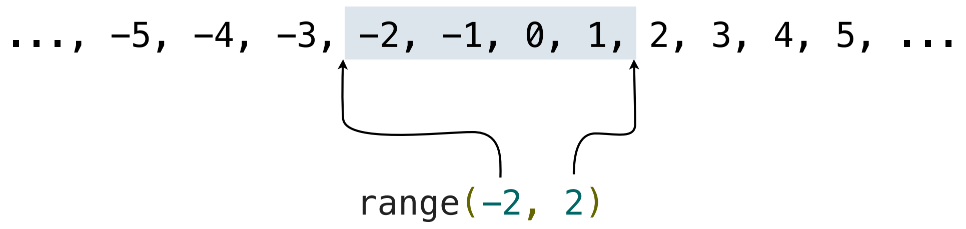
Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)

>>> tuple(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.*



Length: ending value - starting value

Element selection: starting value + index

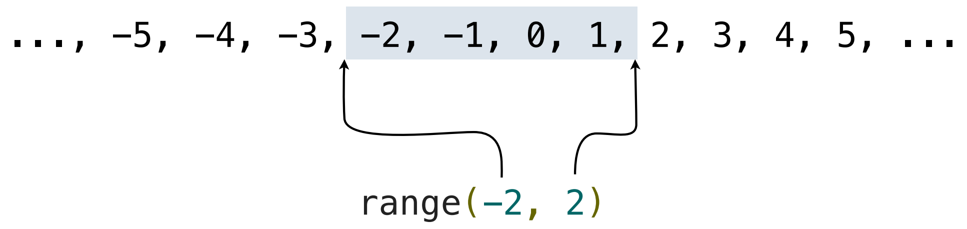
```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)

>>> tuple(range(4))
(0, 1, 2, 3)
```

Tuple constructor

* Ranges can actually represent more general integer sequences.

The Range Type

A range is a sequence of consecutive integers.*



Length: ending value - starting value

Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)
```

Tuple constructor

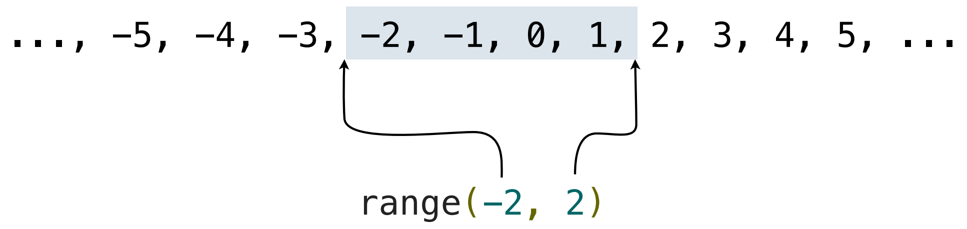
```
>>> tuple(range(4))
(0, 1, 2, 3)
```

With a 0 starting value

* Ranges can actually represent more general integer sequences.

The Range Type

A range is a sequence of consecutive integers.*



Length: ending value - starting value

(Demo)

Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)
```

Tuple constructor

```
>>> tuple(range(4))
(0, 1, 2, 3)
```

With a 0 starting value

* Ranges can actually represent more general integer sequences.

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```
>>> digits[0:2]
(1, 8)
>>> digits[1:]
(8, 2, 8)
```

Membership & Slicing

The Python sequence abstraction has two more behaviors!

Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```
>>> digits[0:2]
(1, 8)
>>> digits[1:]
(8, 2, 8)
```

Slicing creates a new object

Membership & Slicing

The Python sequence abstraction has two more behaviors!

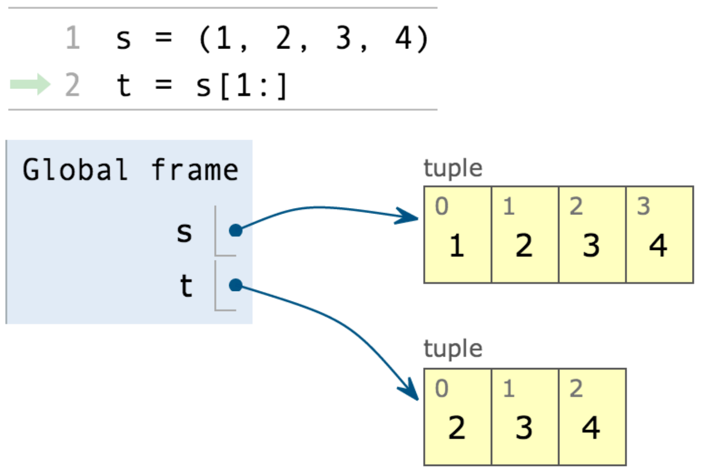
Membership.

```
>>> digits = (1, 8, 2, 8)
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```
>>> digits[0:2]
(1, 8)
>>> digits[1:]
(8, 2, 8)
```

Slicing creates a new object



```
s = (1, 2, 3, 4)
t = s[1:]
```

Lists

['Demo']

http://docs.python.org/py3k/library/stdtypes.html#mutable-sequence-types

List Comprehensions

List Comprehensions

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

List Comprehensions

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

Short version: [<map exp> for <name> in <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:

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 extending the current frame.

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 extending the current frame.
2. Create an empty *result list* that is the value of the expression.

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 extending the current frame.
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 extending the current frame.
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

```
[<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 extending the current frame.
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.
   - B. If `<filter exp>` evaluates to a true value, then add the value of `<map exp>` to the result list.

Dictionaries

```
{'Dem': 0}
```

Limitations on Dictionaries

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs.

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** an object of **a mutable built-in** type.

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** an object of **a mutable built-in** type.
- Two **keys cannot be equal.** There can be at most one value for a given key.

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** an object of **a mutable built-in** 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 **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** an object of **a mutable built-in** 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 an intentional consequence of the dictionary abstraction.

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs.

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** an object of **a mutable built-in** 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 an intentional consequence of the dictionary abstraction.

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

Identity and Equality

(Demo)

Example: http://goo.gl/5AbYNM