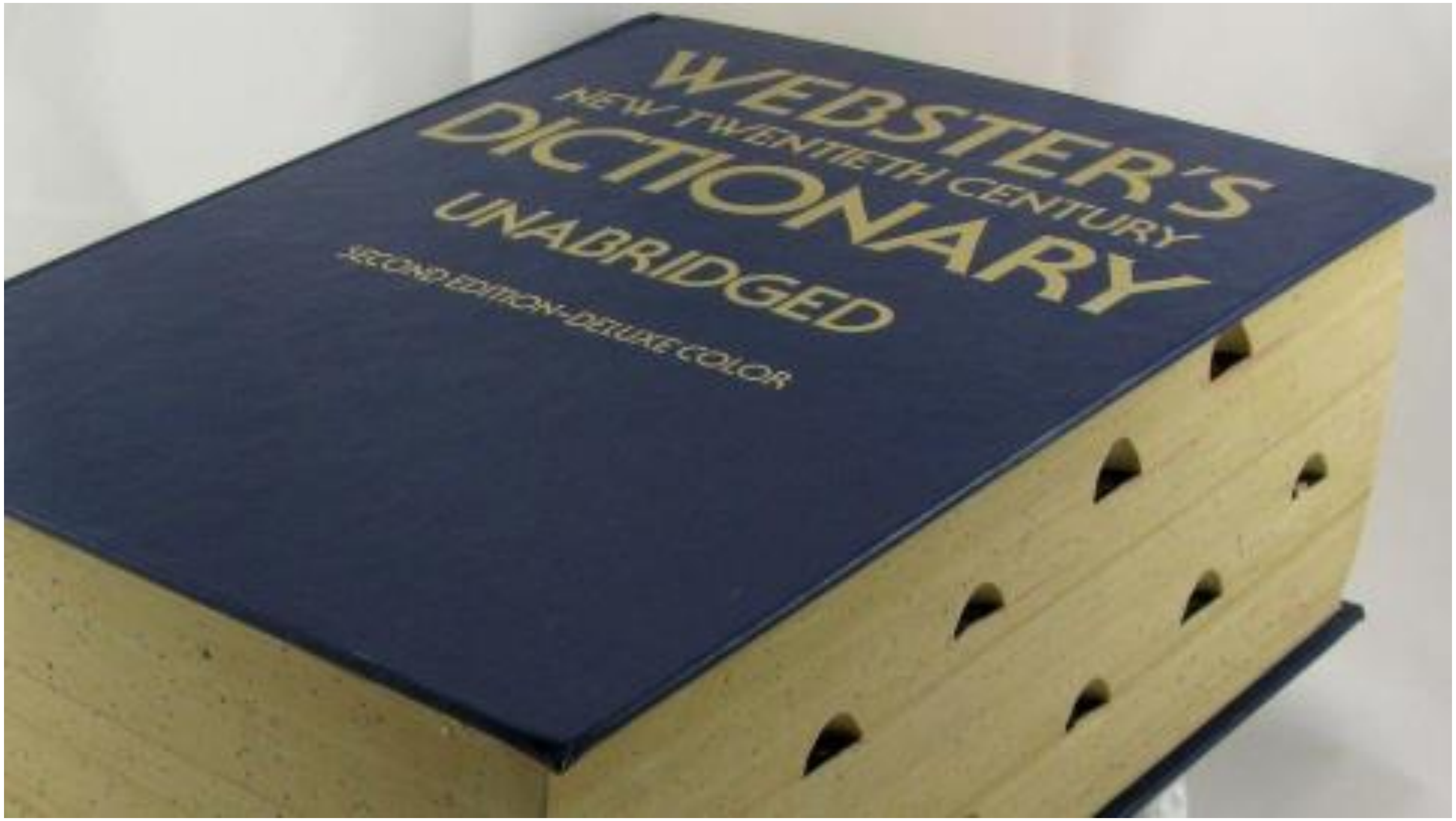


CSCI 141

Lecture 21
Dictionaries

Announcements

Announcements

- Office hours today are half an hour later:
  Usually: 2:00-3:30.
  Today: **2:30-4:00**.

Goals

- Know the basics of how to use dictionaries (`dicts`):
  - Creation, assignment, and indexing
  - `get` method
  - `in` operator
  - `del` statement
  - Iterating over keys and values:
  - `keys`, `values`, and `items` methods

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
A = ["Tony", "Steve"]
B = ("Tony", "Steve")
C = "Tony, Steve"
A[0] = "Thor"
B[0] = "Thor"
print(A[0] + C[:4])
C[0] = "P"
A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
 B = ("Tony", "Steve")
 C = "Tony, Steve"
 A[0] = "Thor"
 B[0] = "Thor"
 print(A[0] + C[:4])
 C[0] = "P"
 A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
✓B = ("Tony", "Steve")
 C = "Tony, Steve"
 A[0] = "Thor"
 B[0] = "Thor"
 print(A[0] + C[:4])
 C[0] = "P"
 A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
✓B = ("Tony", "Steve")
✓C = "Tony, Steve"
 A[0] = "Thor"
 B[0] = "Thor"
 print(A[0] + C[:4])
 C[0] = "P"
 A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
✓B = ("Tony", "Steve")
✓C = "Tony, Steve"
✓A[0] = "Thor"
 B[0] = "Thor"
 print(A[0] + C[:4])
 C[0] = "P"
 A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
✓B = ("Tony", "Steve")
✓C = "Tony, Steve"
✓A[0] = "Thor"
✗B[0] = "Thor"
 print(A[0] + C[:4])
 C[0] = "P"
 A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
✓B = ("Tony", "Steve")
✓C = "Tony, Steve"
✓A[0] = "Thor"
✗B[0] = "Thor"
✓print(A[0] + C[:4])
 C[0] = "P"
 A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
✓B = ("Tony", "Steve")
✓C = "Tony, Steve"
✓A[0] = "Thor"
✗B[0] = "Thor"
✓print(A[0] + C[:4])
✗C[0] = "P"
 A[1:] = ["Bruce", "Natasha"]
```

QOTD

Execute the statements below in order and select the statements that will **not** cause an error. If a statement results in an error, assume it was skipped when executing all statements that follow.

```
✓A = ["Tony", "Steve"]
✓B = ("Tony", "Steve")
✓C = "Tony, Steve"
✓A[0] = "Thor"
✗B[0] = "Thor"
✓print(A[0] + C[:4])
✗C[0] = "P"
✓A[1:] = ["Bruce", "Natasha"]
```

QOTD

What does the following code print?

```python
a = ["Tony", "Steve", "Natasha", "T'Challa", "Carol"]

b = a[2:3] + [a[4]]

b.extend(a[:2])

print(b[2], b[2:3])
```

QOTD

What does the following code print?

```
a = ["Tony", "Steve", "Natasha", "T'Challa", "Carol"]

b = a[2:3] + [a[4]]  # ["Natasha", "Carol"]

b.extend(a[:2]) # ["Natasha", "Carol", "Tony", "Steve"]

print(b[2], b[2:3]) # print("Tony", ["Tony"])
```

```
Tony ['Tony']
```

Last Time: Lists

```
a = [3]
a.append(4)
b = [5, 7]
c = a + b
print(len(a), c[2], b[1])
```

A. 4 5 7

B. 2 5 7

C. 4 7 7

D. 2 5 4

Today: Dictionaries

- Lists, tuples, strings are all **sequences** (their contents are ordered)
- Python also has some types that handle non-sequential collections, including dictionaries (type `dict`):
  - A dictionary is an unordered collection of **key-value mappings**

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

represents the following mapping:

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

represents the following mapping:

0: "B"
1: "A"
2: 7

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

represents the following mapping:

0: "B" → the index `0` maps to the value `"B"`

1: "A"

2: 7

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

`["B", "A", 7]`

represents the following mapping:

0: "B" → the index `0` maps to the value `"B"`

1: "A"

2: 7

A **dictionary** is a **mapping**

from *arbitrary immutable keys*

to *arbitrary values*.

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

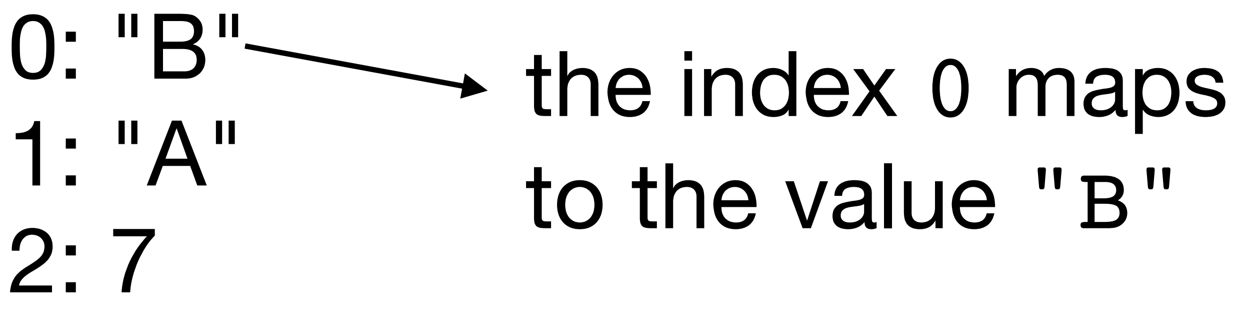
from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

represents the following mapping:



0: "B"
1: "A"
2: 7

the index `0` maps to the value `"B"`

A **dictionary** is a **mapping**

from *arbitrary immutable keys*

to *arbitrary values*.

```
{"B": 6, "A": 7}
```

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

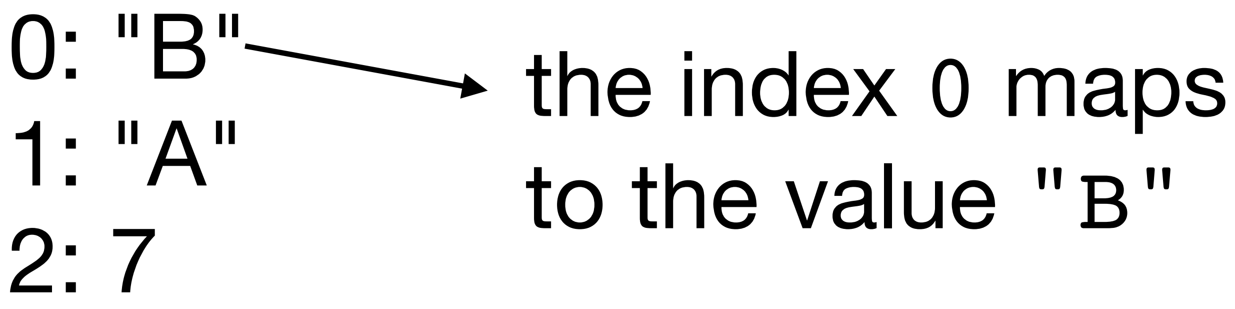
from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

represents the following mapping:



0: "B"
1: "A"
2: 7

the index `0` maps to the value `"B"`

A **dictionary** is a **mapping**

from *arbitrary immutable keys*

to *arbitrary values*.

```
{"B": 6, "A": 7}
```

represents the following mapping:

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

represents the following mapping:

0: "B" → the index `0` maps to the value `"B"`
1: "A"
2: 7

A **dictionary** is a **mapping**

from *arbitrary immutable keys*

to *arbitrary values*.

```
{"B": 6, "A": 7}
```

represents the following mapping:

"B": 6
"A": 7

Dictionaries

Another way to think about **lists**:

A **list** is a **mapping**

from *integer indices*

to *arbitrary values*.

Example:

```
["B", "A", 7]
```

represents the following mapping:

0: "B" → the index `0` maps to the value `"B"`
1: "A"
2: 7

A **dictionary** is a **mapping**

from *arbitrary immutable keys*

to *arbitrary values*.

```
{"B": 6, "A": 7}
```

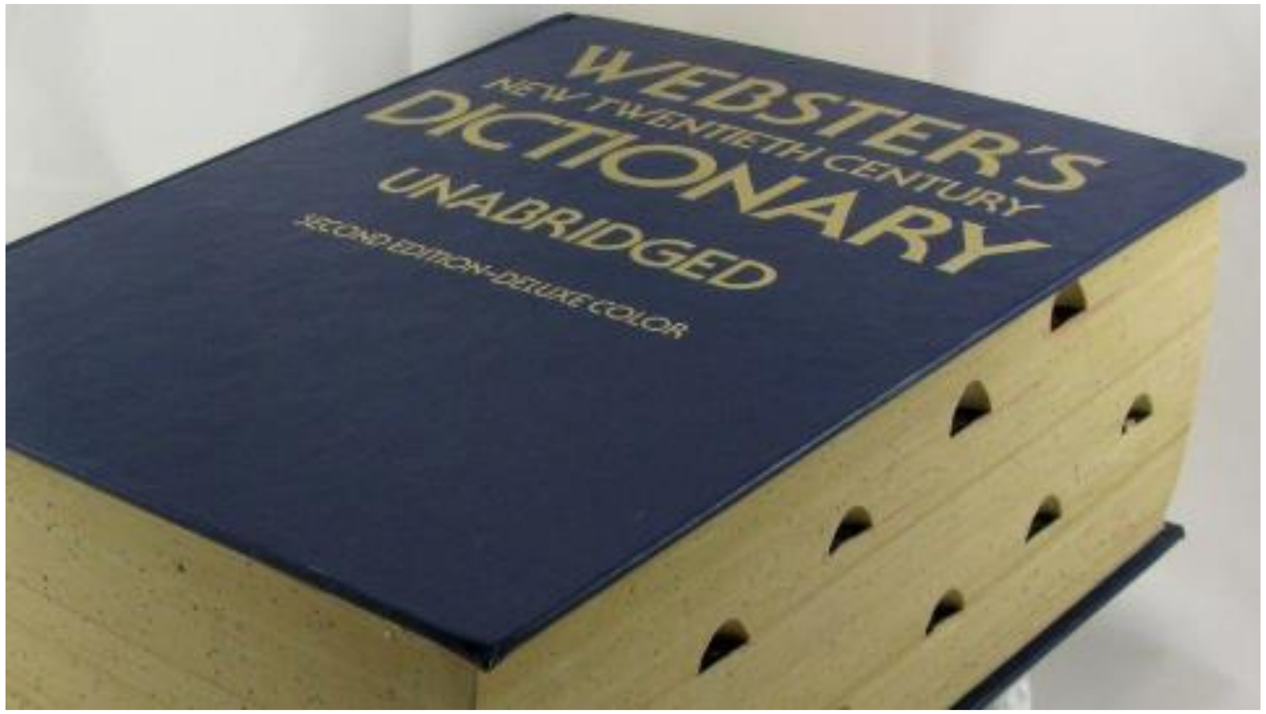
represents the following mapping:

"B": 6 → the key `B` maps to the value `6`
"A": 7

Dictionaries

Why do we want this?

Suppose I want to store...



```
english = {}
english["aardvark"] = """a nocturnal burrowing
mammal with long ears, a tubular snout, and a
long extensible tongue, feeding on ants and
termites. Aardvarks are native to Africa and have
no close relatives."""
```

Dictionaries

Why do we want this?

Suppose I want to store...

A list of W#s of all the students in each of the lab sections.

```
sections = {}
sections[20891] = ["W0183782", "W0243810", # ...
sections[20892] = ["W0184582", "W0182368", # ...
# ...
```

Dictionaries

Why do we want this?

Suppose I want to store...

A bunch of different information about a WWU employee:

```
employee = {"First": "Scott",
            "Last": "Wehrwein",
            "Type": "Faculty",
            "W#": 98765438,
            # ...}
```

Dictionaries

Why do we want this?

Suppose I want to store...

The number of students with each letter grade in my class:

```
grade_counts = {"A": 6, "B": 12, "C": 8, "D": 2}
```

Dictionaries: Let's play

Dictionaries: Let's play

```
# create a dict:
grades = {"A": 10, "B": 18, "C": 6, "D": 2}
grades["A"] # => 10
grades["B"] # => 18
grades["E"] # KeyError
grades["E"] = "Huh?" # new mapping
grades["A"] = 12 # overwrites existing value
"F" in grades # => False
"E" in grades # => True
del grades["E"] # removes "E" and its value
"E" in grades # => False
```

Dictionaries: Let's play

```
# several ways to access values:
grades["A"] # => 12
grades.get("A") # => 12

# get method never causes an error
grades["Q"] # KeyError
grades.get("Q") # => None (no error!)

# get can take a default value to
# return if the key isn't found:
grades.get("A", 0) # => 12
grades.get("Q", 0) # => 0
```

Dictionaries: Cheat Sheet

if key exists: overwrite old value

otherwise: add new key-value mapping

Dictionaries: Cheat Sheet

- Creation:

```
d = {key1: value1, key2: value2, ...}
```

if key exists: overwrite old value

otherwise: add new key-value mapping

Dictionaries: Cheat Sheet

- Creation:

  `d = {`*`key1`*`: `*`value1`*`, `*`key2`*`: `*`value2`*`, ...}`

- Access:

  `d[`*`key`*`] # => `*`value`*, or **error** if key not in d

  `d.get(`*`key`*`) # => `*`value`*, *or* ***`None`*** *if key not in d*

  `d.get(`*`key`*`, `*`alt`*`) # => `*`value`*, *or* ***`alt`*** *if key not in d*

if key exists: overwrite old value

otherwise: add new key-value mapping

Dictionaries: Cheat Sheet

- Creation:
  d = {*key1*: *value1*, *key2*: *value2*, ...}

- Access:
  d[*key*] # => *value*, or **error** if key not in d
  d.get(*key*) # => *value*, *or* ***None*** *if key not in d*
  d.get(*key*, *alt*) # => *value*, *or* ***alt*** *if key not in d*

- Assignment:
  d[*key*] = *new_value*
  → if key exists: overwrite old value
  → otherwise: add new key-value mapping

Dictionaries: Cheat Sheet

- Creation:

  d = {*key1*: *value1*, *key2*: *value2*, ...}

- Access:

  d[*key*] # => *value*, or **error** if key not in d

  d.get(*key*) # => *value, or* ***None*** *if key not in d*

  d.get(*key*, *alt*) # => *value, or* ***alt*** *if key not in d*

- Assignment:

  d[*key*] = *new_value*

if key exists: overwrite old value

otherwise: add new key-value mapping

- Membership:

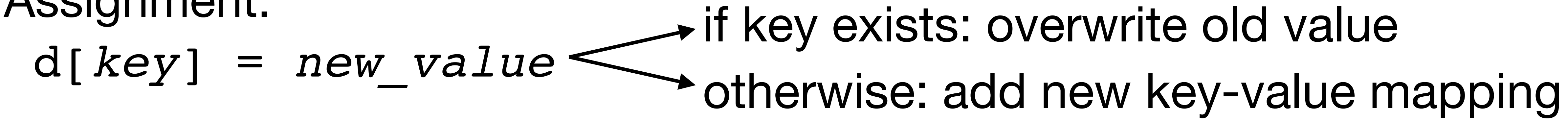
  key in d # => True if d[*key*] exists

Dictionaries: Cheat Sheet

- Creation:
  `d = {`*`key1`*`: `*`value1`*`, `*`key2`*`: `*`value2`*`, ...}`

- Access:
  `d[`*`key`*`] # => `*`value`*, or **error** if key not in d
  `d.get(`*`key`*`) # => `*`value`*, *or* ***`None`*** *if key not in d*
  `d.get(`*`key`*`, `*`alt`*`) # => `*`value`*, *or* ***`alt`*** *if key not in d*

- Assignment:



  `d[`*`key`*`] = `*`new_value`*

  if key exists: overwrite old value

  otherwise: add new key-value mapping

- Membership:
  `key`` in d # => True` if `d[`*`key`*`]` exists

- Removal:
  `del d[`*`key`*`] # deletes key and its associated value`

Worksheet - Problem 1

```
def charcount(in_string):
    """ Return a dictionary that maps each unique character in
    in_string to the number of times it appears in the string.
    Precondition: in_string is a string
    Example: count("hahah") # => {"a": 3, "h": 2} """
```

- Creation:
  `d = {key1: value1, key2: value2, ...}`

- Access:
  `d[key] # => value`, or **error** if key not in d
  `d.get(key) # => value`, *or* ***None*** *if key not in d*
  `d.get(key, alt) # => value`, *or* ***alt*** *if key not in d*

- Assignment:
  `d[key] = new_value`

- Membership:
  `key in d # => True` if `d[key]` exists

Iterating over Dictionaries? Demo

```
pop = {"WWU": 16121, "UW": 47899, "WSU": 24470}
```

Iterating over Dictionaries? Demo

```
pop = {"WWU": 16121, "UW": 47899, "WSU": 24470}
```

- for key in d
- d.keys(); list(d.keys())
- for val in d.values()
- key, value in d.items()
- list(d.items())

Dictionaries: Iterating

```
d = {key1: value1, key2: value2, ...}

for key in d:
    print(key)

for key in d.keys():
    print(key)

for val in d.values():
    print(val)

for (key, val) in d.items():
    print(key, val, sep=": ")
```

Dictionaries: Iterating

```
d = {key1: value1, key2: value2, ...}

for key in d:
    print(key)

for key in d.keys():
    print(key)

for val in d.values():
    print(val)

for (key, val) in d.items():
    print(key, val, sep=": ")
```

Note 1: Like `range`, these methods return sequences that are not lists. To get a list of values use `list(d.values())`.

Dictionaries: Iterating

```
d = {key1: value1, key2: value2, ...}

for key in d:
    print(key)

for key in d.keys():
    print(key)

for val in d.values():
    print(val)

for (key, val) in d.items():
    print(key, val, sep=": ")
```

Note 1: Like `range`, these methods return sequences that are not lists. To get a list of values use `list(d.values())`.

Note 2: You **can't** rely on iteration happening in any particular order!

Worksheet - Exercise 2

```
def strmode(in_str):
    """ Return the most frequently-appearing character in
    in_str, or any of the most frequent characters in case of a
    tie. Precondition: in_str is a string with nonzero length.
    Examples: strmode('hahah') # => 'h'
              strmode('who') # => could return 'w', 'h', or 'o'
    """
```

- Creation:
  d = {*key1*: *value1*, *key2*: *value2*, ...}

- Access:
  d[*key*] # => *value*, or **error** if key not in d
  d.get(*key*) # => *value, or **None** if key not in d*
  d.get(*key*, *alt*) # => *value, or **alt** if key not in d*

- Assignment:
  d[*key*] = *new_value*

- Membership:
  key in d # => True if d[*key*] exists

Worksheet - Exercise 2

```
def strmode(in_str):
    """ Return the most frequently-appearing character in
    in_str, or any of the most frequent characters in case of a
    tie. Precondition: in_str is a string with nonzero length.
    Examples: strmode('hahah') # => 'h'
              strmode('who') # => could return 'w', 'h', or 'o'
    """
```

- Creation:
  `d = {`*`key1`*`: `*`value1`*`, `*`key2`*`: `*`value2`*`, ...}`
- Access:
  `d[`*`key`*`] # => `*`value`*, or **error** if key not in d
  `d.get(`*`key`*`) # => `*`value`*, *or **None** if key not in d*
  `d.get(`*`key`*`, `*`alt`*`) # => `*`value`*, *or **alt** if key not in d*
- Assignment:
  `d[`*`key`*`] = `*`new_value`*
- Membership:
  `key`` in d # => True` if `d[`*`key`*`]` exists

Hint: use your `charcount` function, then find the **key** whose **value** is largest.