CSCI 141

Lecture 22

References and Functions

Lists and Dictionaries: methods and manipulations

Happenings

Tuesday, 5/28 – CS Poster Session!
– 3 – 5 pm in the 4th Floor Hallway!

Tuesday, 5/28 – ACM Research Talk: Machine Learning with Dr. Hutchinson
– 5 pm in CF 316

Tuesday, 5/28 – Peer Lecture Series: Machine Learning Workshop
– 5 pm in CF 165

Tuesday, 5/28 – Artificial Intelligence Presents: Machine Learning!
– 6 pm in PH 228

Thursday 5/30 – CS Picnic!
– 4 – 7 pm at the Lake Padden Playground Picnic Shelter!

Announcements

Announcements

- A5 Code and A5 Written are out.

Announcements

- A5 Code and A5 Written are out.
- No class or office hours Monday

Announcements

- A5 Code and A5 Written are out.
- No class or office hours Monday
- No lab next week

Announcements

- A5 Code and A5 Written are out.
- No class or office hours Monday
- No lab next week
- Office hours Tuesday from 12ish to 3

A5 Code

xkcd.com/1838

Goals

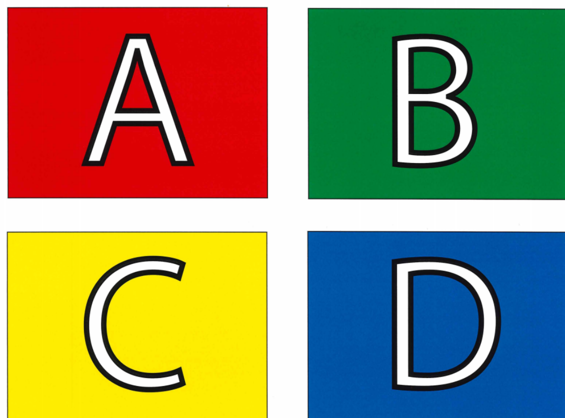
- Understand the implications of variables holding references to mutable objects:
    - function parameters can refer to objects that are also referred to by global variables
- Know how to modify lists using the following: `insert, remove, del`
- Know the basics of how to use dictionaries (dicts):
    - Creation, assignment, indexing
    - in, del, iterating over keys and values

Last Time

- Understand the implications of variables holding references to mutable objects: *multiple* variables can refer to the *same object*

- Know how to modify lists using indexed assignment, slice assignment

What does the code below print?



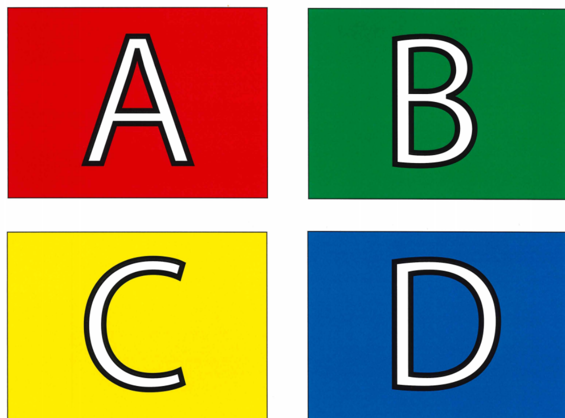

```
a = [1, 2, 3]
b = a
b[2] = 1
print(a[1:])
```

Last Time

- Understand the implications of variables holding references to mutable objects: *multiple* variables can refer to the *same object*

- Know how to modify lists using indexed assignment, slice assignment

What does the code below print?



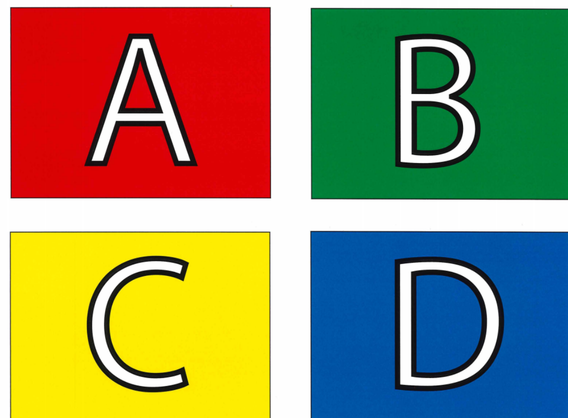

```
a = [1, 2, 3]
b = a
b[2] = 1
print(a[1:])
```

```
A.  [1, 2, 3]
B.  [2, 3]
C.  [1, 2, 1]
D.  [2, 1]
```

Last Time

Nuance: what if we assign a slice instead?

What does the code below print?

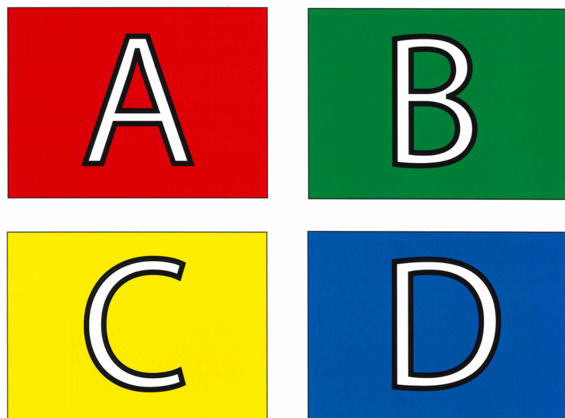



```
a = [1, 2, 3]
b = a[1:3]
b[1] = 1
print(a[1:])
```

Last Time

Nuance: what if we assign a slice instead?

What does the code below print?



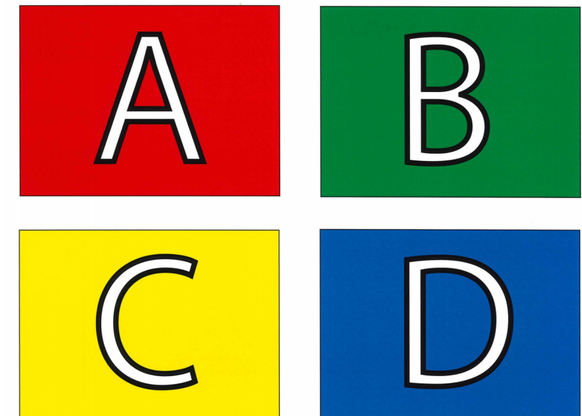

```
a = [1, 2, 3]
b = a[1:3]
b[1] = 1
print(a[1:])
```

A. [1, 2, 3]
B. [2, 3]
C. [1, 2, 1]
D. [2, 1]

Last Time

- Know the basics of how to use dictionaries (`dicts`):
  - Creation, assignment, indexing

What does the code below print?



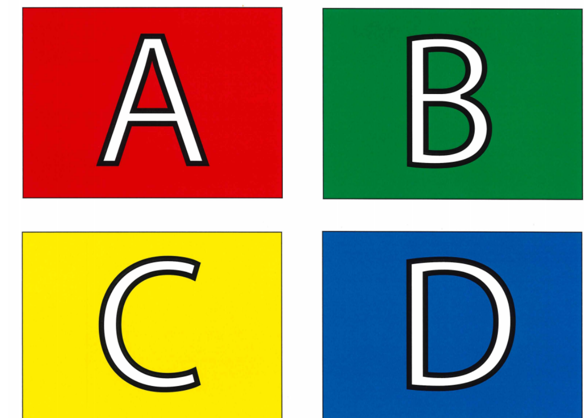

```
gc = {"A": 8, "B": 12, "C": 6}
gc["A"] += 1
gc["C"] -= 1
gc["D"] = 1
print(gc["C"] + gc["D"])
```

Last Time

- Know the basics of how to use dictionaries (`dicts`):
  - Creation, assignment, indexing

What does the code below print?

```
gc = {"A": 8, "B": 12, "C": 6}
gc["A"] += 1
gc["C"] -= 1
gc["D"] = 1
print(gc["C"] + gc["D"])
```

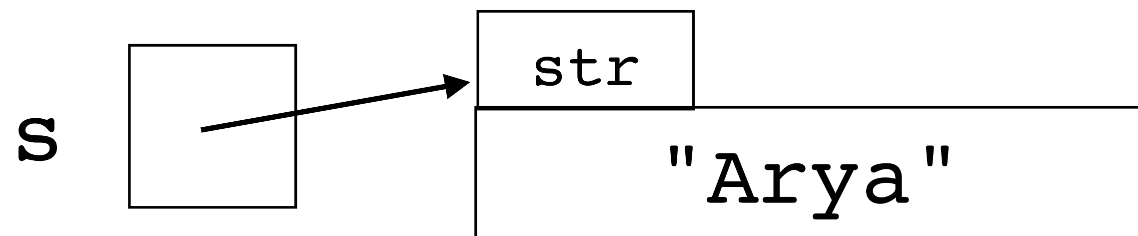


A. 5
B. 6
C. 7
D. error

Back to Mutability and Functions

- Lists and dictionaries are mutable: you can change their contents.
- Strings, tuples, ints, and floats, are immutable: you can't change their value.

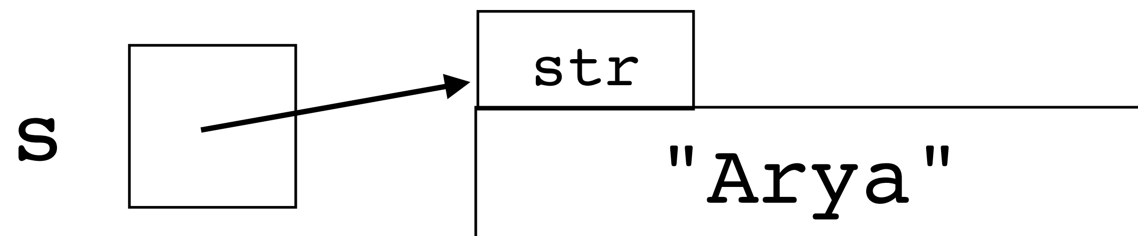
```
s = "Arya"
```



Back to Mutability and Functions

- Lists and dictionaries are mutable: you can change their contents.
- Strings, tuples, ints, and floats, are immutable: you can't change their value.

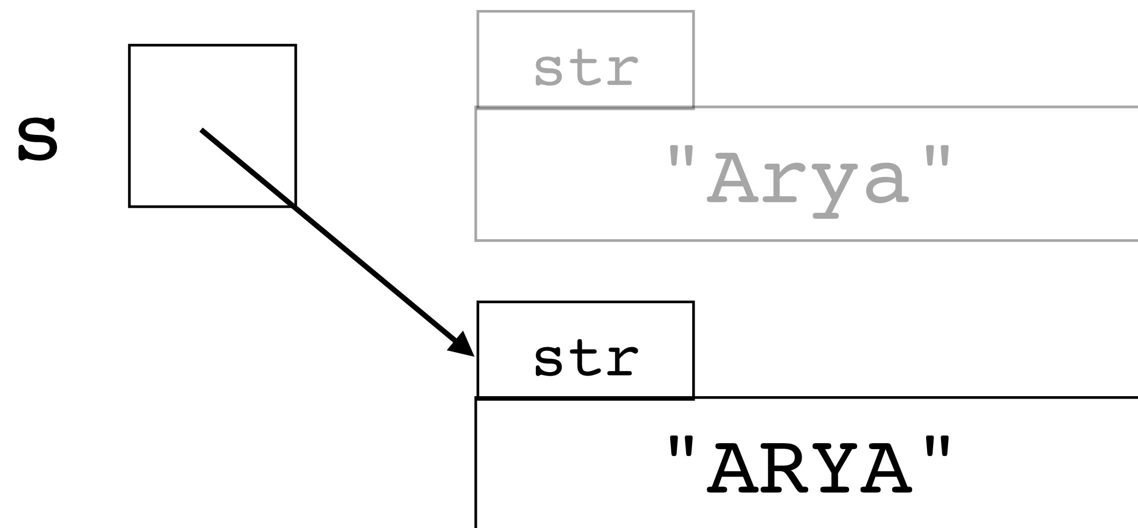
```
s = "Arya"
s = s.upper()
```



Back to Mutability and Functions

- Lists and dictionaries are mutable: you can change their contents.
- Strings, tuples, ints, and floats, are immutable: you can't change their value.

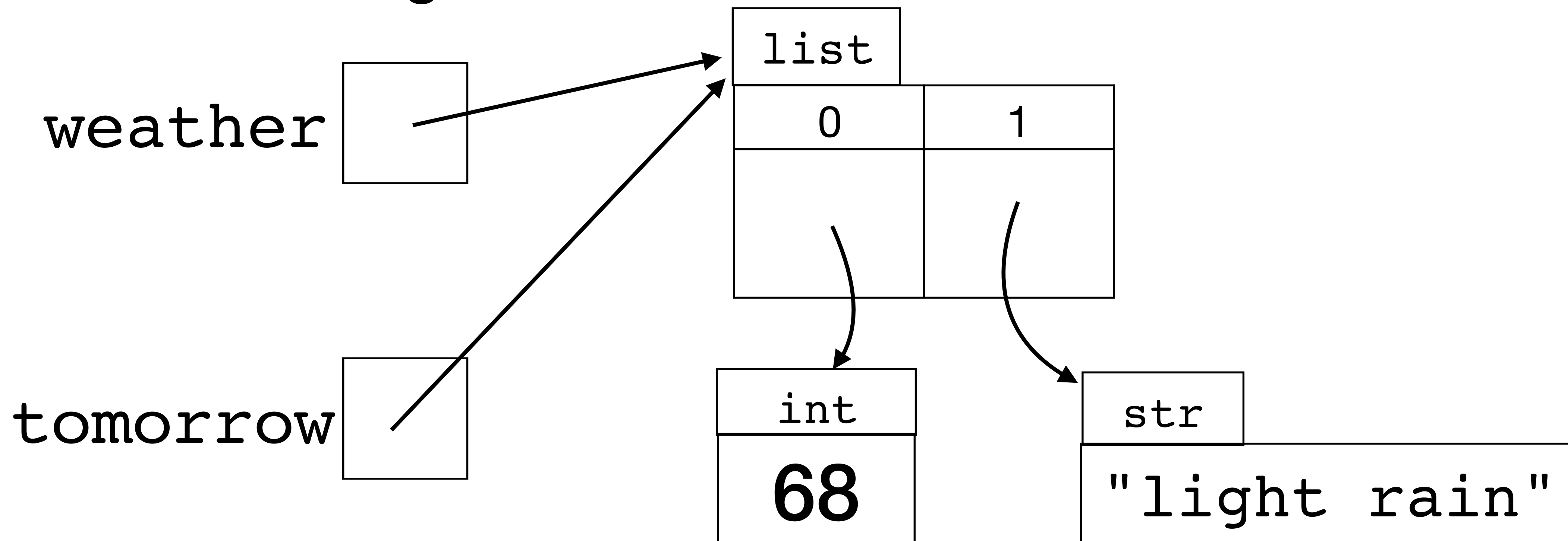
```
s = "Arya"
s = s.upper()
```



Implications of Mutability

```
weather = [63, "light rain"]
tomorrow = weather
tomorrow[0] = 68
print(weather[0])
```

After creating the initial list:



Implications of Mutability

```
weather = [63, "light rain"]
tomorrow = weather
tomorrow[0] = 68
print(weather[0])
```

Implications of Mutability

```
weather = [63, "light rain"]
tomorrow = weather
tomorrow[0] = 68
print(weather[0])
```

More than one variable can refer to the **same object**.

Implications of Mutability

```
weather = [63, "light rain"]
tomorrow = weather
tomorrow[0] = 68
print(weather[0])
```

More than one variable can refer to the **same object**.

Changes to an object via one variable are reflected when accessing it via another variable!

Implications of Mutability

```
weather = [63, "light rain"]
tomorrow = weather
tomorrow[0] = 68
print(weather[0])
```

More than one variable can refer to the **same object**.

Changes to an object via one variable are reflected when accessing it via another variable!

To create a true copy of a **mutable** object, you can't simply assign a new variable to the object.

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a *reference* to the list:

Mutable Objects and Functions

(or any mutable object!)

When you pass a list into a function, you're actually passing a *reference* to the list:

Mutable Objects and Functions

(or any mutable object!)

When you pass a list into a function, you're actually passing a *reference* to the list:

```
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a *reference* to the list:

```
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a *reference* to the list:

```
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```

`a_list` points to the **same** list as the global variable `a`

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

The local variable `a_list` is reassigned to point to a **new** (different) list

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

The local variable `a_list` is reassigned to point to a **new** (different) list

The list referenced by `a` is **unchanged**.

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```
def z3(x):
    a_list = [x, x, x]
    return a_list
b = 2
a = z3(b)
print(a)
```

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```
def z3(x):
    a_list = [x, x, x]
    return a_list
b = 2
a = z3(b)
print(a)
```

The function creates a **new** list, with the local variable `a_list` referring to it.

Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```
def z3(x):
    a_list = [x, x, x]
    return a_list
b = 2
a = z3(b)
print(a)
```

The function creates a **new** list, with the local variable `a_list` referring to it.

The **reference** to the list is returned and assigned to `a`.

Worksheet - Exercise 1

Write a function that returns a true copy (i.e., a different list object containing the same values).

```
def copy_list(in_list):
    """ Return a new list object containing
        the same elements as in_list.
        Precondition: in_list's contents are
        all immutable. """
```

Worksheet - Exercise 1

Write a function that returns a true copy (i.e., a different list object containing the same values).

```
def copy_list(in_list):
    """ Return a new list object containing
        the same elements as in_list.
        Precondition: in_list's contents are
        all immutable. """
```

Hint: one possible approach uses a loop and the `append` method.

Worksheet - Exercise 1

Write a function that returns a true copy (i.e., a different list object containing the same values).

```
def copy_list(in_list):
    """ Return a new list object containing
        the same elements as in_list.
        Precondition: in_list's contents are
        all immutable. """
```

Hint: one possible approach uses a loop and the `append` method.

When done, complete Exercise 1A: Draw the memory diagram for the following code snippet:

```
a = [1, 2]
b = copy_list(a)
b[0] = 0
```

A few more list operations:

A few more list operations:

```
my_list.index(value)
```

Return the index of value in my_list
Throw an error if value is not in my_list.

A few more list operations:

```
my_list.index(value)
```

Return the index of value in my_list
Throw an error if value is not in my_list.

```
my_list.insert(index, value)
```

Inserts value into my_list at index, shifting all following elements on spot to the right.

A few more list operations:

```
my_list.index(value)
```

Return the index of value in my_list
Throw an error if value is not in my_list.

```
my_list.insert(index, value)
```

Inserts value into my_list at index, shifting all following elements on spot to the right.

```
my_list.remove(value)
```

Inserts value into my_list at index, shifting all following elements one spot to the right.

A few more list operations:

```
my_list.index(value)
```

Return the index of value in my_list
Throw an error if value is not in my_list.

```
my_list.insert(index, value)
```

Inserts value into my_list at index, shifting all following elements on spot to the right.

```
my_list.remove(value)
```

Inserts value into my_list at index, shifting all following elements one spot to the right.

```
del my_list[index]
```

Removes the element at index, shifting all following elements one spot to the left.

index, insert, remove, del: Demo

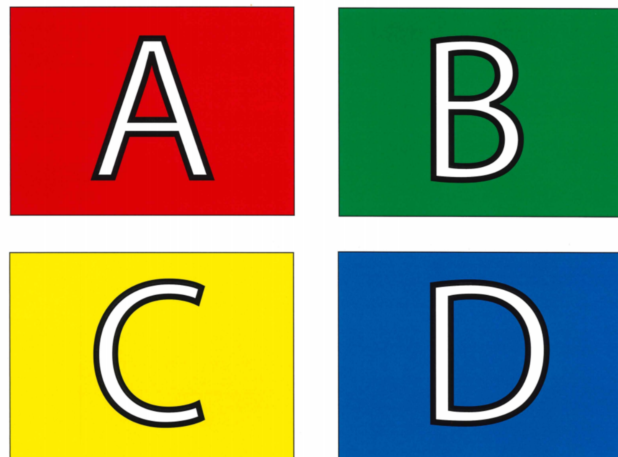
index, insert, remove, del: Demo

```
abc = ["B", "C"]
abc.index("C")
abc.index("F")
abc.insert(0, "A")
abc.remove("C")
abc.remove("F")
del abc[0]
```

```
b = []
a.insert(0, b)
b[0] = 4
a.insert(0, 4]
```

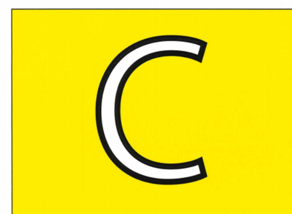
What does this print?

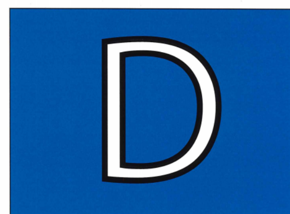
```
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
print(a)
```



What does this print?

```
a = []
b = [1]
a.insert(0, b)
b[0] = 4
a.insert(0, b)
print(a)
```





A. [1, 4]
B. [4, 4]
C. [[1], [4]]
D. [[4], [4]]

Dictionaries: TL;DR

Dictionaries: TL;DR

- Creation:

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

Dictionaries: TL;DR

- 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*

Dictionaries: TL;DR

- 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`*

Dictionaries: TL;DR

- 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

Dictionaries: TL;DR

- 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

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

Worksheet - Exercise 2

```python
def count(values):
    """ Return a dictionary that maps each element of values to
        the number of times it appears in the list.
        Precondition: values is a list of immutable objects """
```

- 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

Dictionaries: Iterating

Dictionaries: Iterating

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

Dictionaries: Iterating

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

for key in d:
    print(key)
```

Dictionaries: Iterating

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

for key in d:
    print(key)

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

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)
```

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

Worksheet - Exercise 3

```
def mode(values):
    """ Return the most frequently-appearing value in values,
        or one of the most frequent values in case of a tie.
        Precondition: values is a list of immutable objects
    """
```

Worksheet - Exercise 3

```
def mode(values):
    """ Return the most frequently-appearing value in values,
        or one of the most frequent values in case of a tie.
        Precondition: values is a list of immutable objects
    """
```

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