Lecture 8: Lists and Mutability

Check-in and Reminders

- Reminder: **Homework 3 out:** pick up from the front
- How to approach homework questions:
    - You can test out pieces of code in interactive python
    - But the best way to learn is to walk through the logic of the code using pencil and paper (without a machine)
        - Homework are the best practice for exams!
- Lab 3 due tonight for Mon labs, tomorrow night for Tues
- Our office hours
    - Today: Iris (12-1 pm), Me (12.30 - 2pm @ CS common room)
    - Tomorrow: Iris (10 am-noon), Me (1-2 pm)

Do You Have Any Questions?

Fast Paced Course: Practice is Key!

- This is a very fast paced course
- To keep up, **you must practice what we learn in lectures**
- **Learning a new language is all about immersing yourself in it**
- Best way to learn French?
    - Go live in France for a bit
- Best way learn Python?
    - **Live in PythonWorld!** Play with examples in interactive python
    - Test out code we do in class on your own
- **Get ahead, stay ahead.** Preparing for the lab by reviewing lectures will make you more productive!

Do You Have Any Questions?

Review: Lists

- We have worked with lists as a sequence (ordered collection of items)
- We know how to concatenate two lists with a +
- We know how to append an item to a list
- Lists, unlike strings, are a mutable sequence
- This means we can update them
    - Add items to lists
    - Delete items from lists
    - Sort lists in place, etc
- Today we will discuss lists in more detail and implications of lists being mutable

Updating by Reassignment

- Update by direct assignment to a list index

Example.

```
myList[1] = 7  # reassign to an existing index
```

`myList` Before

`[1, 2, 3, 4]`

`myList` After

`[1, 7, 3, 4]`

Append()

`myList.append(item)` : appends item to end of list

Example.

```
myList.append(5)  # stick 5 at the end of the list
```

`myList` Before

```
[1, 7, 3, 4]
```

`myList` After

```
[1, 7, 3, 4, 5]
```

Extend()

`myList.extend([itemList])`: appends all the items in itemList to the end of myList

Example.

```
myList.extend([6, 8]) # stick both 6 and 8 at the
end of the list
```

`myList` Before	`myList` After
`[1, 7, 3, 4, 5]`	`[1, 7, 3, 4, 5, 6, 8]`

Pop()

`myList.pop(index):` Removes the item at a given index **and returns it.** If no index is given, removes and returns the last item from the list.

Example.

`myList.pop(3)` returns `4`

`myList` Before

`[1, 7, 3, 4, 5, 6, 8]`

`myList` After

`[1, 7, 3, 5, 6, 8]`

Pop()

`myList.pop(index):` Removes the item at a given index **and returns it.** If no index is given, removes and returns the last item from the list.

Example.

`myList.pop()` → returns → `8`

No Index

`myList` Before	`myList` After
`[1, 7, 3, 5, 6, 8]`	`[1, 7, 3, 5, 6]`

Insert()

`myList.insert(index, item):` inserts item at index in myList, all items to the right of index shift over to make room

Example.

```
myList.insert(0,11)  # insert 11 at index 0
```

`myList` Before	`myList` After
`[1, 7, 3, 5, 6]`	`[11, 1, 7, 3, 5, 6]`

Insert()

`myList.insert(index, item):` inserts item at index in myList, all items to the right of index shift over to make room

Example.

inserting at an index out of range

```
myList.insert(10,12)  # insert 12 at index 10
```

`myList` Before	`myList` After
`[11, 1, 7, 3, 5, 6]`	`[11, 1, 7, 3, 5, 6, 12]`

Remove()

`myList.remove(item):` removes item from myList, all items to the right removed item shift to the left by one

Example.

```
myList.remove(12)    # remove 12 from myList
```

`myList` Before

```
[11, 1, 7, 3, 5, 6, 12]
```

`myList` After

```
[11, 1, 7, 3, 5, 6]
```

Sort()

`myList.sort(item):` sorts the list in place in ascending order

Example.

```
myList.sort()    # sort by mutating myList
```

`myList` Before

```
[11, 1, 7, 3, 5, 6]
```

`myList` After

```
[1, 3, 5, 6, 7, 11]
```

Sort() vs Sorted()

- Sort method is only for lists and sorted by mutating the list itself (it does not return anything!)
- Sorted can be used for any sequence (strings, lists, tuples), **it returns** a new sorted sequence, and does NOT modify the original sequence

Example.

```
list1 = [6, 3, 4],  list2 = [6, 3, 4]
list1.sort() # sort by mutating list1
sorted(list2) # returns a new sorted list
```

`list1` Before	`list1` After	`list2` Before	`list1` After
[6, 3, 4]	[3, 4, 6]	[6, 3, 4]	[6, 3, 4]

Does not change!

Value vs Identity

- An objects **identify** never changes in Python once it has been created, you may think of it as the object's address in memory
- The `is` operator compares the identity of two objects, the `id()` function returns an integer representing its identity
- The **value** of some objects can change. Objects whose values can change are called **mutable;** objects whose values cannot change are called **immutable**
- The == operator compares the value (contents) of an object
- **Question.** Which mutable objects have you encountered so far?

Mutability in Python

Strings, Ints, Floats are Immutable

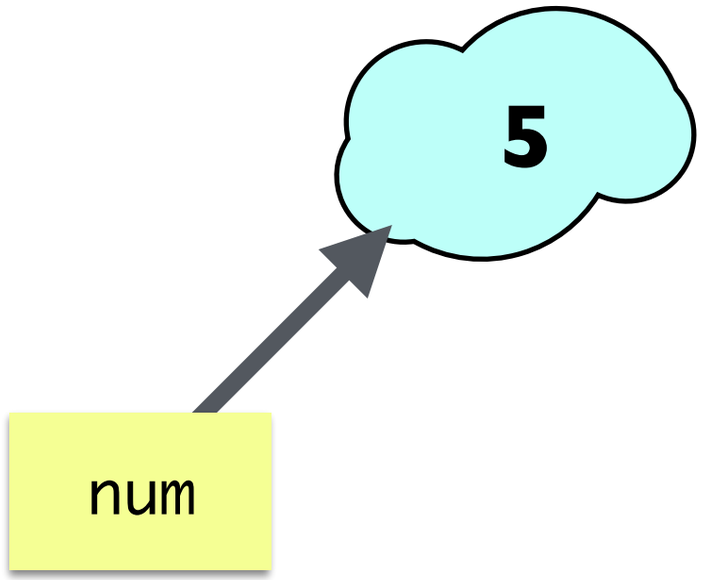
- Once you create them, their value cannot be changed!
- **All functions that we have seen on these return a new object and do not modify the original object**

Lists are Mutable

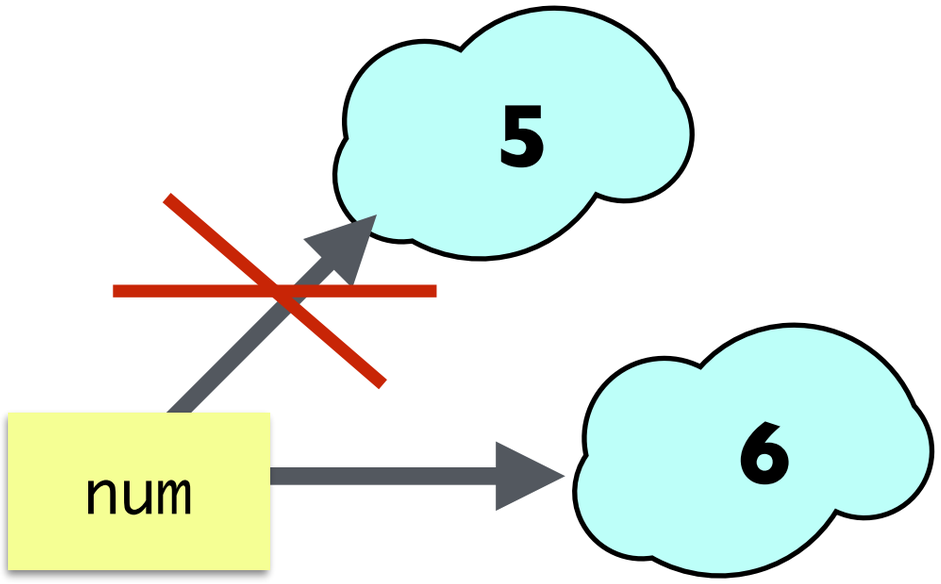
- Lists are mutable sequences
- As we saw, you can mutate what's in a list in many ways
- Mutability of lists has many implications such as aliasing, which can cause more trouble than its worth if we are not careful!

Mutability in Python

```
>>> num = 5
```



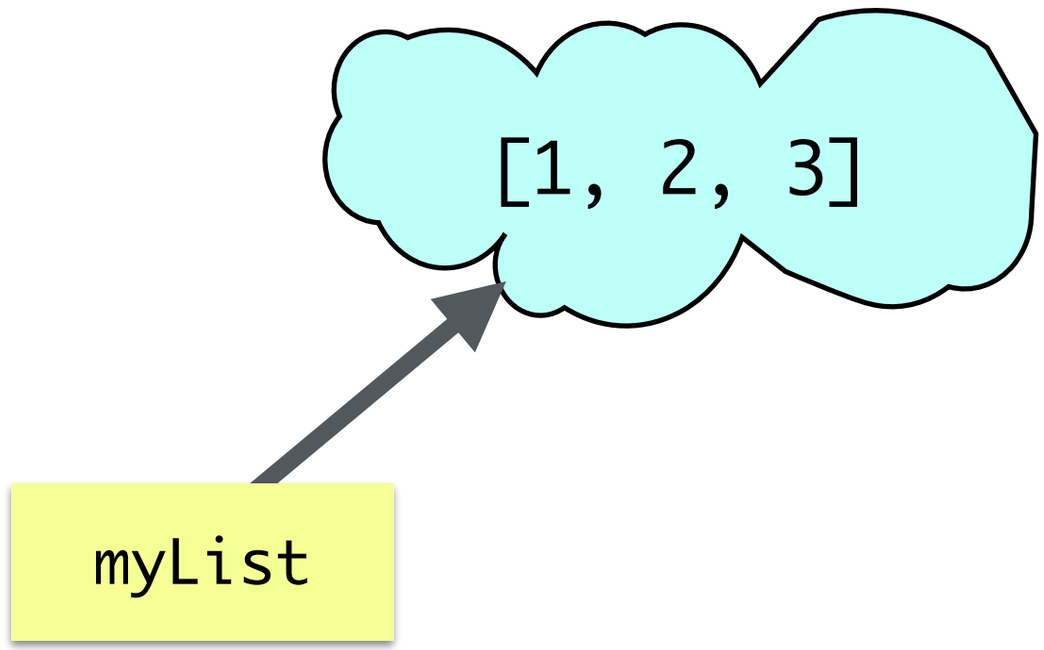

```
>>> num = num + 1
```



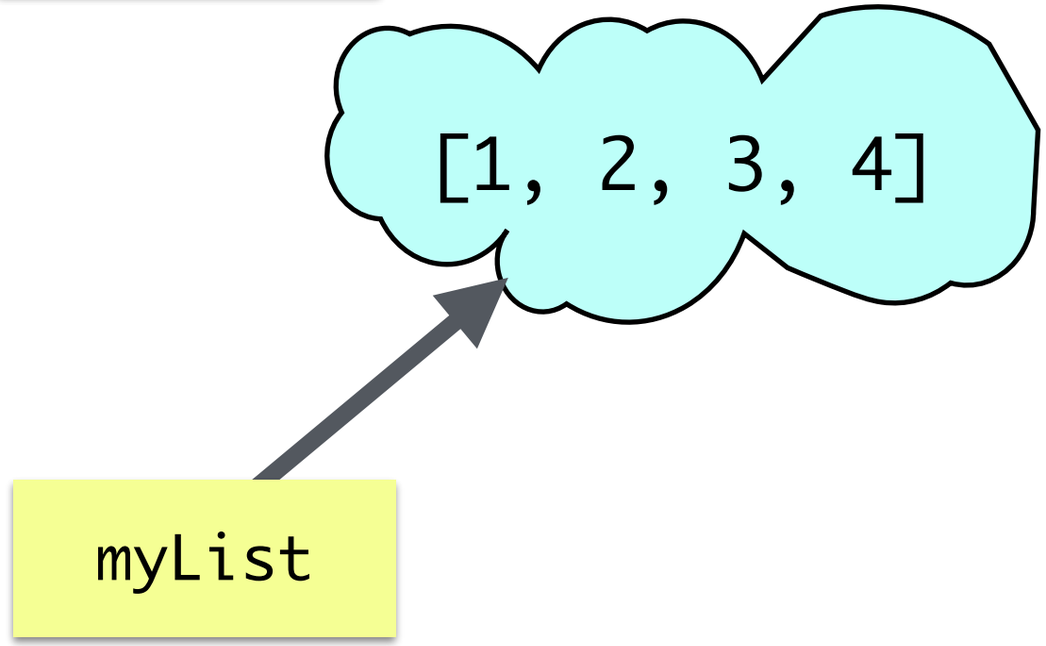
Strings, Ints, Floats are Immutable

Mutability in Python

```
>>> myList = [1, 2, 3]
```




```
>>> myList.append(4)
```



Lists are Mutable

Mutability in Python

```
>>> word = 'Williams'
>>> college = word
>>> word == college
True
>>> word is college
True
```

Even though word and college have the same identity now, if we tried to update one of them it would just assume a new identity!

Strings are Immutable

```
>>> myList = [1, 2, 3]
>>> newList = [1, 2, 3]
>>> list2 = myList
>>> myList == newList
True
>>> myList is newList
False
>>> myList == list2
True
>>> myList is list2
True
```

Lists are Mutable

List Aliasing

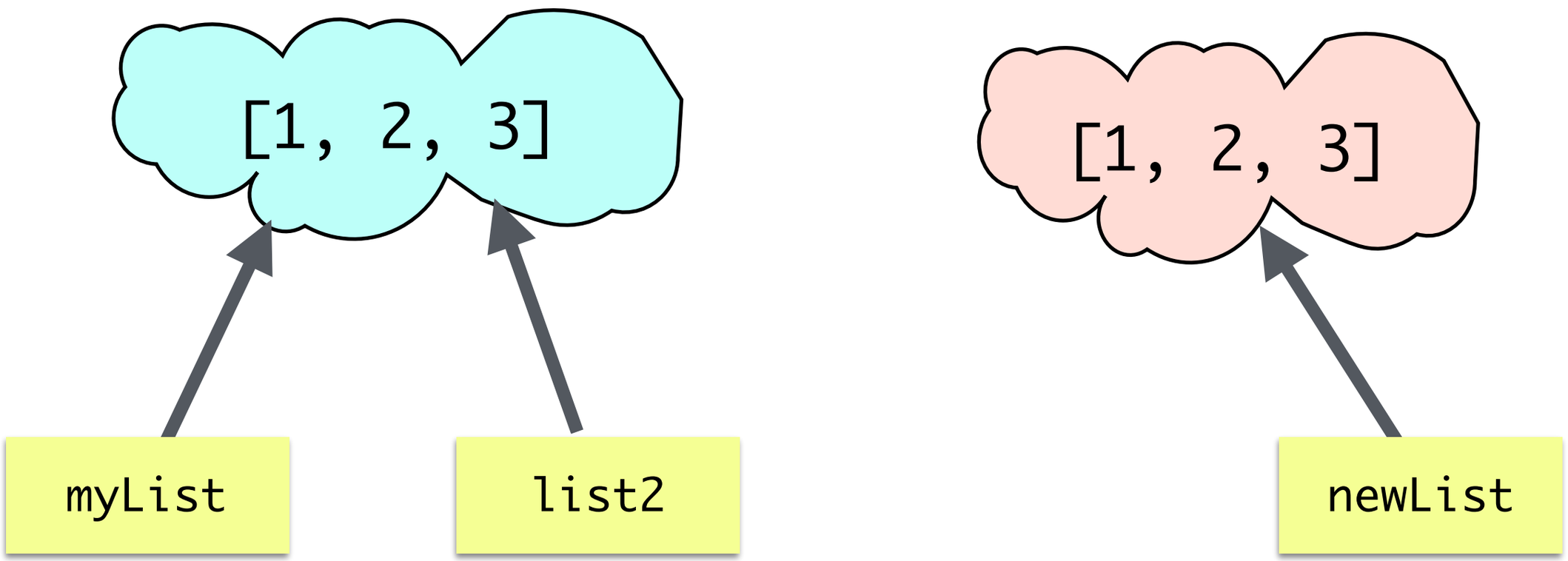
- Any assignment or operation that “points” to a list implicitly creates an alias

```
>>> myList = [1, 2, 3]
>>> list2 = myList                # creates an alias!
>>> newList = [1, 2, 3]
>>> list2 is myList
```

returns → `True`

```
>>> myList is newList
```

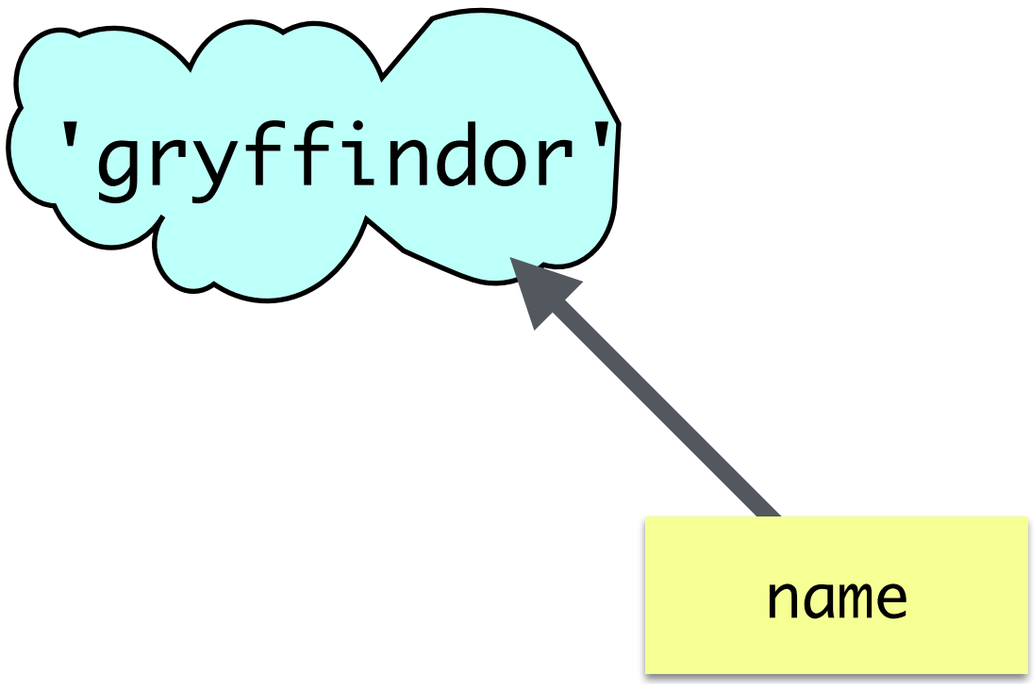
returns → `False`



Int, floats, Str are NOT mutable

- Int, str and float are immutable, once created they can never be changed. Any operation on them creates **a new object.**

```
name = 'gryffindor'
```

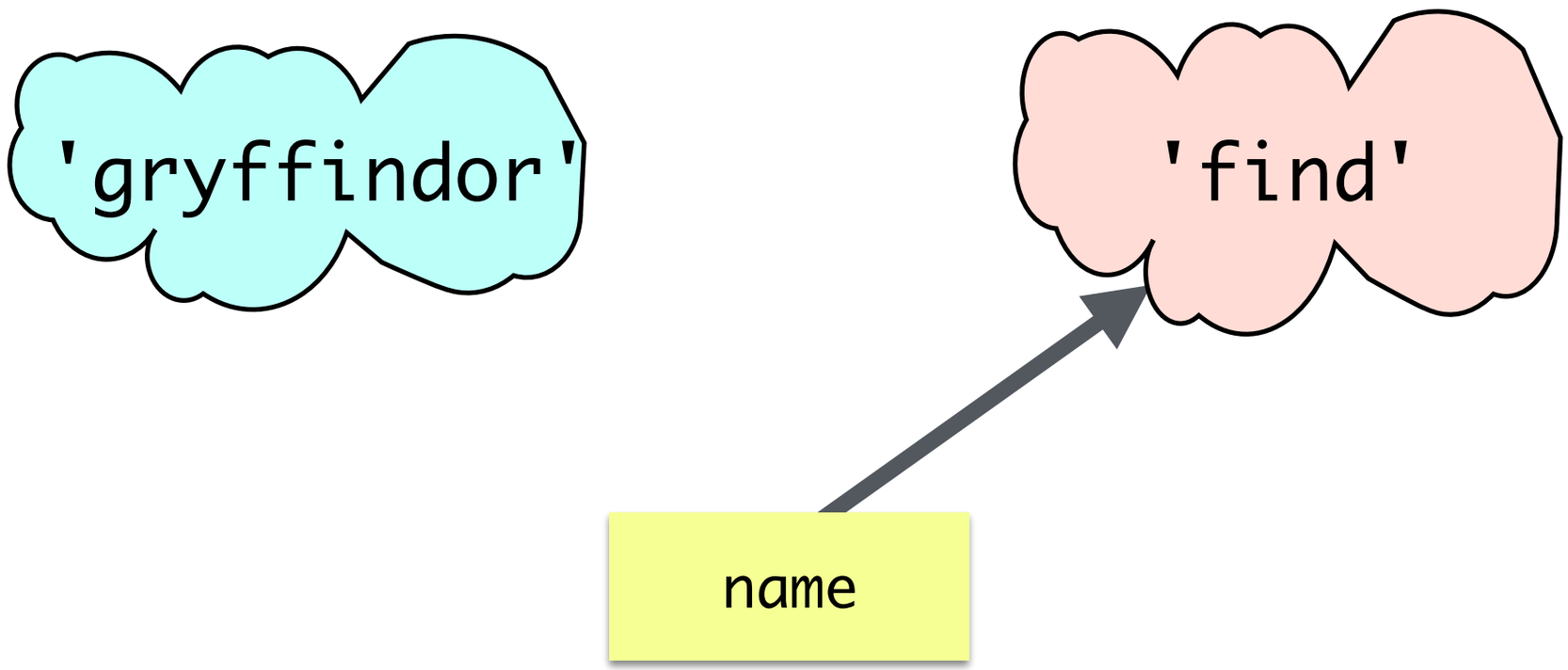


Int, floats, Str are NOT mutable

- Int, str and float are immutable, once created they can never be changed. Any operation on them creates **a new object.**

```
name = 'gryffindor'

name = name[4:8]   # returns a new string, gets assigned to name
```



Seq Operations: Return a new Seq

- The following operations that can be performed on both lists and strings always return a new list/string
    - sorted(sequence): returns a **new** sorted sequence
    - slicing operator: returns a **new** sliced sequence
    - assignment of a **new sequence** to a variable

```
word = 'Shikha'

myList = [1, 2, 3]
```

- concatenation always creates a new sequence
- operations like `len`, accessing an element using an index do not modify the sequence

Mutability Quiz: Test Yourself

- Can you explain this?

```
In [68]: a = [15, 20]
         b = [15, 20]
         c = [10, a, b]
         b[1] = 5
         c[1][0] = c[0]
```

```
In [69]: print(a)
```

```
[10, 20]
```

```
In [70]: print(b)
```

```
[15, 5]
```

```
In [71]: print(c)
```

```
[10, [10, 20], [15, 5]]
```

Mutability Quiz: Test Yourself

- Can you explain this?

```
In [76]: a = [15, 20]
         c = [10]
         c.append(a)
         a[1] = 5

In [77]: print(a)

         [15, 5]

In [78]: print(c)

         [10, [15, 5]]
```

Tuples: New Immutable Sequence

Examples:

```
# A homogeneous tuple of five integers
numTup = (5, 8, 7, 1, 3)

# A homogeneous tuple with 4 strings
houseTup = ('Gryffindor', 'Hufflepuff', 'Ravenclaw', 'Slytherin')

# A pair is a tuple with two elements
pair = (7, 3)

# A tuple with one element must use a comma
# to avoid confusion with parenthesized expression
singleton = (7, )

# A tuple with 0 values
emptyTup = ( )

# A tuple without parens, not good practice
noParen = 'a',
```

Tuples: New Immutable Sequence

- Tuples are an immutable sequence of values separated by commas and enclosed within parenthesis ()
- Tuples support any sequence operation that don't involve mutation: e.g., len(), indexing, slicing, concatenation, sorted
- Tuples support simple and nifty assignment

```
harryInfo = ['Harry Potter', 11, True]

name, age, glasses = harryInfo  #tuple assignment!

# is just concise way of writing:

# name = harryInfo[0]

# age = harryInfo[1]

# glasses = harryInfo[2])
```

Format Printing in Python

- A quick way to build strings with particular form is to use the `.format` function on them

Syntax: `myString.format(*args)`

`*args` means it takes zero or more arguments

- For every pair of braces (`{}`), format consumes one argument.
- Argument is converted to a string (with `str`) and concatenated with the remaining parts of the format string
- Especially useful in printing: called **format printing**

```
In [8]: "Hello, you {} world{}".format("silly",'!')  # creates a new string
Out[8]: 'Hello, you silly world!'

In [9]: print("Hello, {}.".format("you silly world!"))
        Hello, you silly world!.
```

Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/