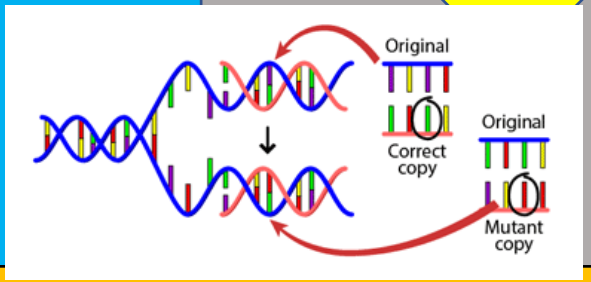
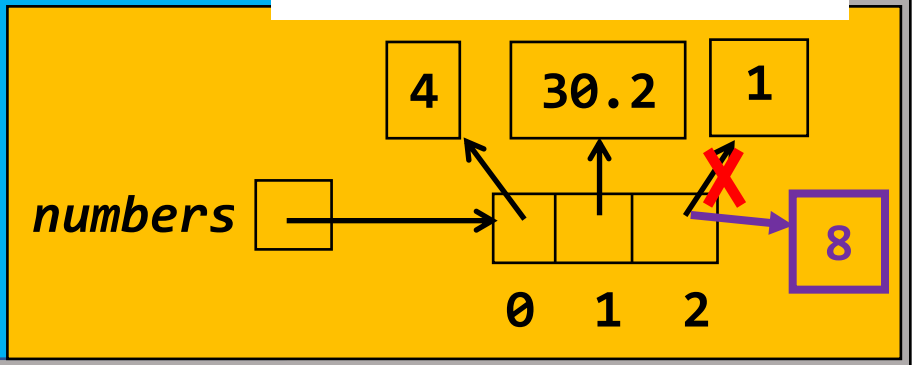
Mutation

1. Review: Objects Names
   Assignment References
   Box-and-pointer diagrams
2. What is *mutation*?
3. Mutable vs. immutable objects
4. Why is mutation *dangerous?*
5. Why is mutation *useful?*
6. Writing code that mutates objects





Review:

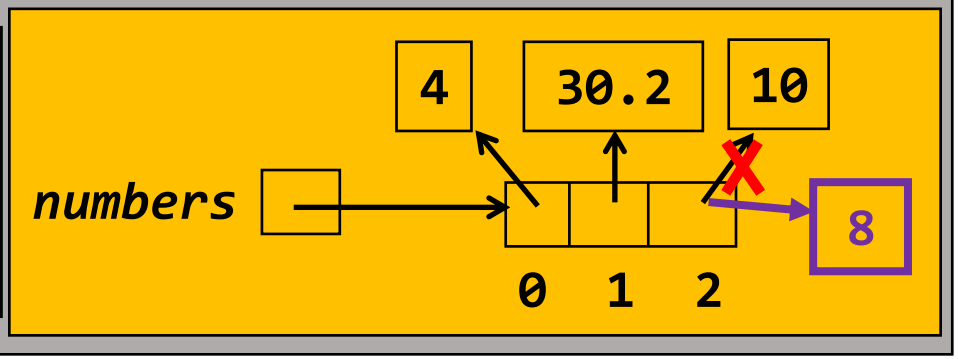
Objects Names

Assignment References

Box-and-pointer diagrams

Next 7 minutes or so are from a video for a session earlier in the course.

```
numbers = [4, 30.2, 10]
numbers[2] = 8
```



Assignment: The statement `blah = value` ***assigns*** the name on the left to the value on the right.

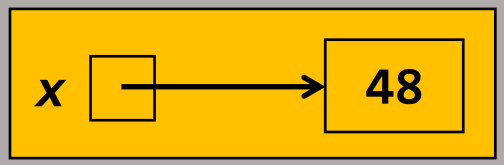
In the 1st line of code to the right, ***x*** is ***assigned*** to the value 48.

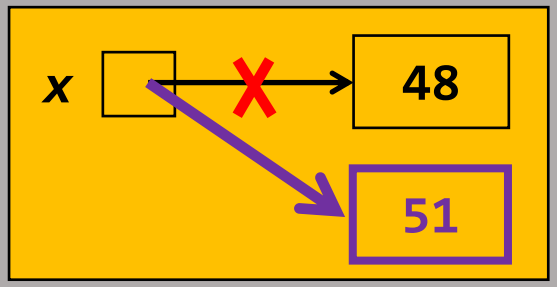
That is, the name (variable) ***x*** is a ***reference*** to the object ***48***.

In the 2nd line of code to the right, ***x*** is ***re-assigned*** to the value ***51***.

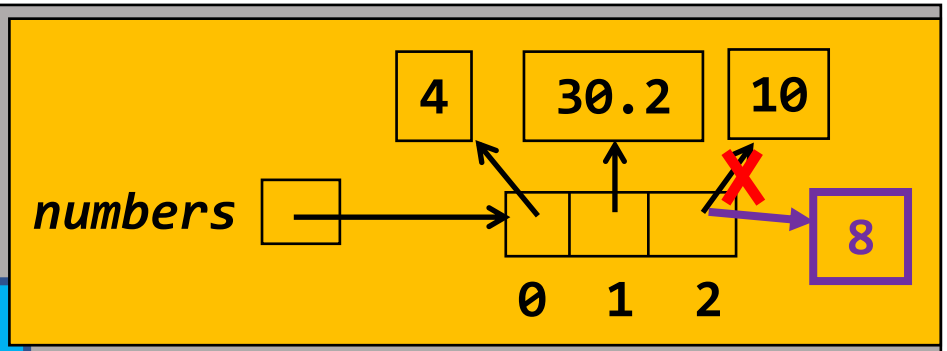
That is, the name (variable) ***x*** is now a ***reference*** to the object ***51***.

```
x = 48
x = x + 3
```





Assignment vs ***Mutation***



For non-container objects, like integers and floating-point numbers, everything works as you would expect.

But when we consider ***container objects*** (objects with INSIDES), we have the possibility that the INSIDES of the container-object is re-assigned. *In this case, the container object is NOT reassigned, but it IS "affected".*

We need a word for what happens to the container object when its insides are re-assigned: We say that **the container object is MUTATED**.

Mutation: The statement

*insides_of*_**blah = value**

assigns the name on the left to the value on the right (as always), but ***mutates*** the name (variable) **blah**.

Mutation: The statement *insides_of*_**blah = value** ***assigns*** the name on the left to the value on the right (as always) but ***mutates*** the name (variable) **blah**.

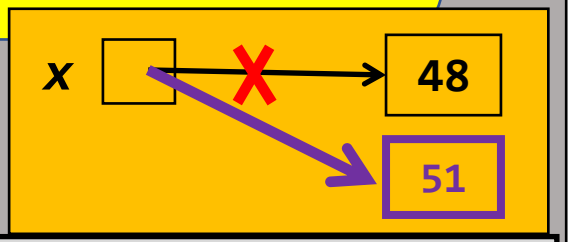
Just as in the examples with **x = 3**:

Here ***numbers*** is ***assigned*** to a ***list*** with three items: **4**, **30.2** and **10**.

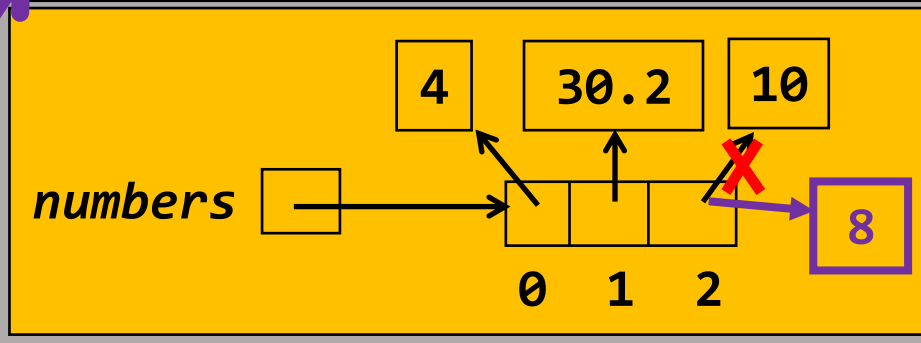
That is, the name (variable) ***numbers*** is a ***reference*** to the object ***[4, 30.2, 10]***.

Here ***numbers[2]*** is ***re-assigned*** to the value **15**.

```
x = 48
x = x + 3
```




```
numbers = [4, 30.2, 10]
numbers[2] = 8
```

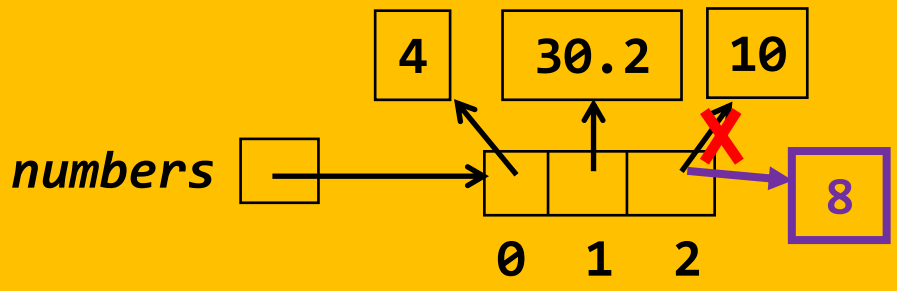


The line of code **`numbers[2] = 8`** ***re-assigns*** `numbers[2]` to 8.

The name (variable) **`numbers`** was NOT re-assigned.

It still refers to the same place in memory, as the picture shows.

```
numbers = [4, 30.2, 10]
numbers[2] = 8
```



But ***numbers*** was "affected" by the re-assignment of its INSIDES. We need a name for that effect.

We say that **`numbers`** was ***MUTATED***.

Only container objects can be mutated.

When an item *inside* a container object is *re-assigned*, we say that the *container object* itself is *mutated*.

Examples

```
numbers = [4, 30.2, 10]
numbers[2] = 8
```

The 2nd line of the code to the left:

- Re-assigns `numbers[2]`
- ***Mutates*** `numbers`

```
r = 20
p = Point(32, 15)
c = Circle(p, r)
c.radius = 40
c.center.y = 99
```

The 4th line of the code to the left:

- Re-assigns `c.radius`
- ***Mutates*** **`c`**

The 5th line of the code to the left:

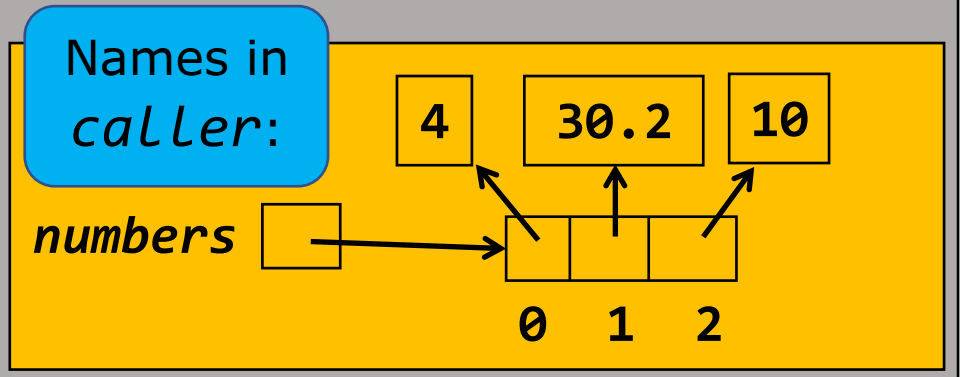
- Re-assigns `c.center.y`
- ***Mutates*** **`c.center`** *and* **`c`**
- ***Mutates*** **`p`**

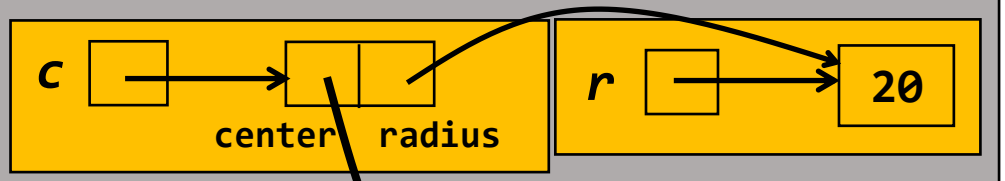
(assuming that the Circle's `__init__` stores a reference to p, NOT to a clone of p)

Function calls do assignments, hence can mutate arguments.

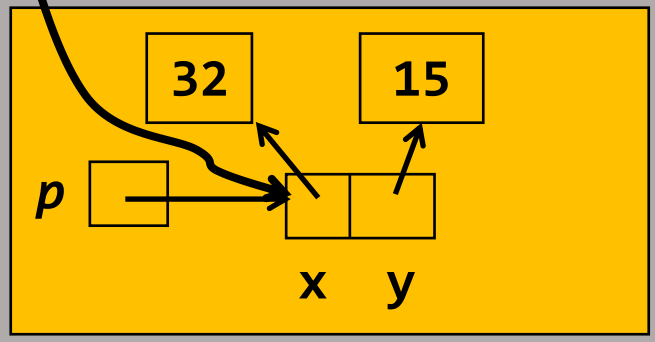
```
def caller():
    numbers = [4, 30.2, 10]
    r = 20
    p = Point(32, 15)
    c = Circle(p, r)
    blah(numbers, c, r)

def blah(nums, c2, rad):
    nums[2] = 8
    c2.radius = 40
    c2.center.y = 99
    rad = 7
```

Names in *caller*:





Assignments in ***caller*** yield this box-and-pointer diagram.

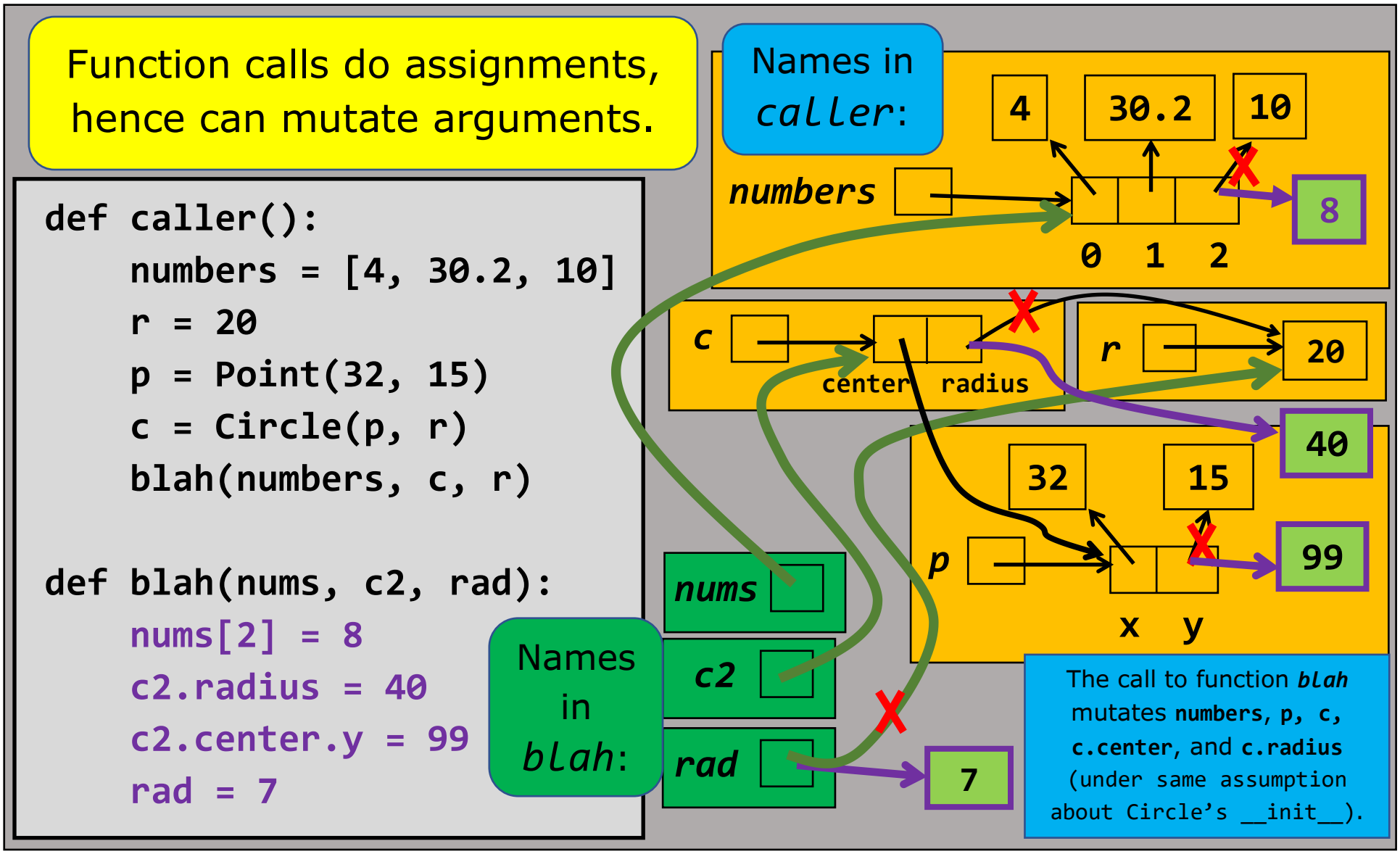


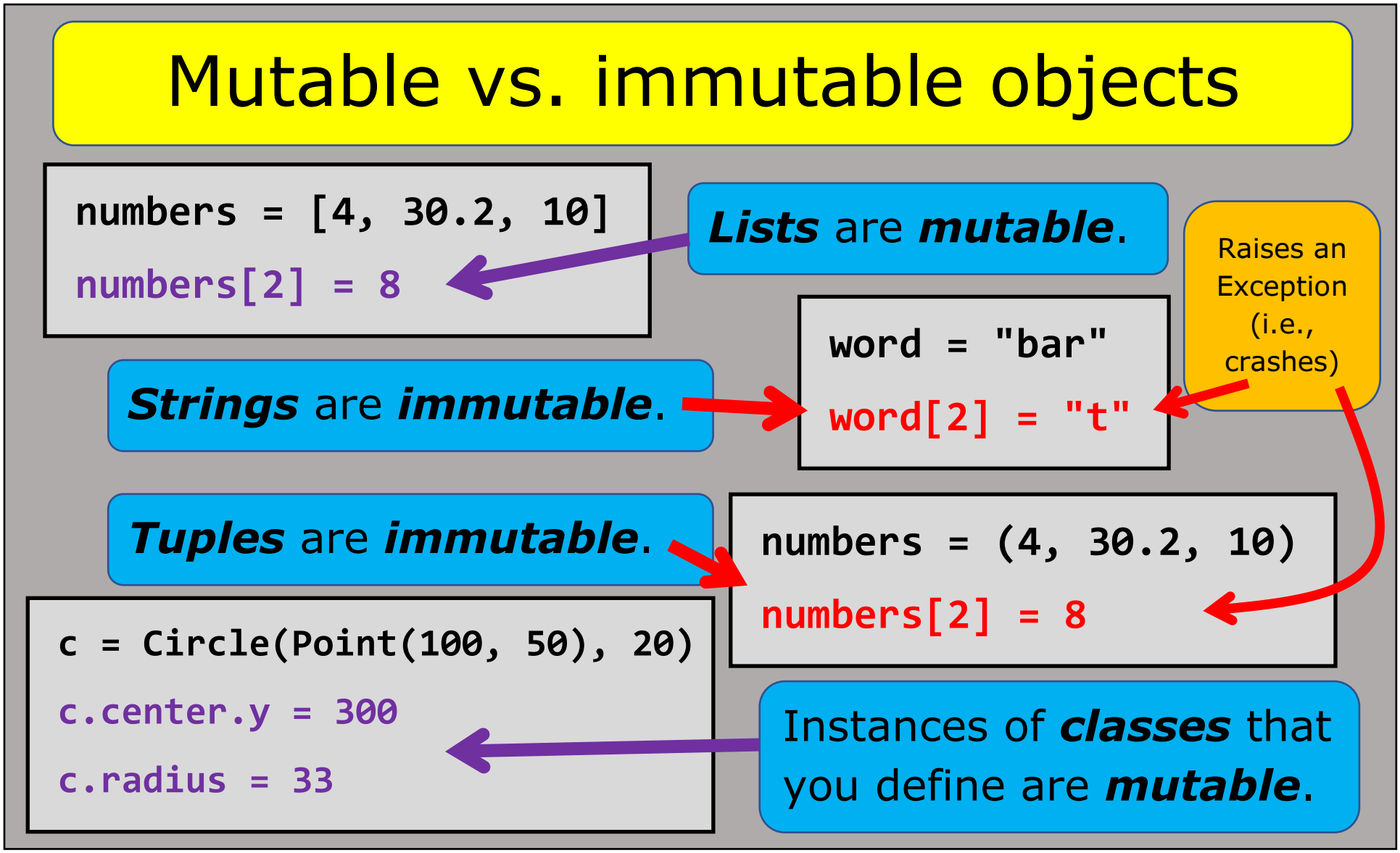
The call to ***blah*** causes (in effect):

`nums = numbers` `c2 = c` `rad = r`

Function calls do assignments, hence can mutate arguments.

```
def caller():
    numbers = [4, 30.2, 10]
    r = 20
    p = Point(32, 15)
    c = Circle(p, r)
    blah(numbers, c, r)

def blah(nums, c2, rad):
    nums[2] = 8
    c2.radius = 40
    c2.center.y = 99
    rad = 7
```




Mutable vs. immutable objects
numbers = [4, 30.2, 10]
numbers[2] = 8
Lists are mutable.
Raises an Exception (i.e., crashes)
word = "bar"
word[2] = "t"
Strings are immutable.
Tuples are immutable.
numbers = (4, 30.2, 10)
numbers[2] = 8
c = Circle(Point(100, 50), 20)
c.center.y = 300
c.radius = 33
Instances of classes that you define are mutable.

Why is mutation dangerous?

```
def caller():
    wallet = Wallet(...)

    # Expects the function to return
    # the number of dollars in the wallet.
    dollars = count_money(wallet)

    # Later on, surprised to find no money!

def count_money(wallet):
    count = # Code that counts the dollars
    # Code that maliciously re-assigns the
    # INSIDES of the wallet to be empty!
    return count
```

Often, when mutation happens there are multiple names for a single object (as per a function call). This can lead to coding errors – mutating the object by one name necessarily mutates the (same) object by the other name(s). Sometimes coders have a blind spot

Worse still, it poses something of a security risk:

Suppose that someone else implemented a function that you call (and foolishly trust).

If you send that function a mutable object, and if the function maliciously mutates the object (unbeknownst to you), that could be a disaster!

Why is mutation useful?

Lots of programs need a function that takes a (big) list as an argument and modifies a few items in the list. Such functions could either:

1. ***Mutate*** the list at the few places it needs to be mutated, or
2. Make a ***copy*** of the list, **mutate the copy**, and ***return*** **the copy**.

Choice #1 is in some sense "safer" –
the caller's list is not changed in any way.

Choice #2 saves both space and time.
If the list is big, the savings can be VERY substantive.

See the Python examples in your repository, especially the one that shows how big the savings can be.

Writing code that mutates objects

See these Python examples in your repository.

Which examples ***mutate*** the argument? Which ***return*** a mutated ***copy*** of the argument?

```
def example1a(numbers):
    for k in range(len(numbers)):
        numbers[k] = numbers[k] + 1


def example1b(numbers):
    new = []
    for k in range(len(numbers)):
        new.append(numbers[k] + 1)
    return new
```

```
def example2a(numbers):
    if len(numbers) > 0:
        numbers[-1] = numbers[-1] + 1


def example2b(numbers):
    new = []
    for k in range(len(numbers)):
        new.append(numbers[k])
    new[-1] = new[-1] + 1
    return new
```

Could we have used
new = new + [numbers[k]]
here? Which version is better? Why?