



UC Berkeley EECS  
Lecturer  
Michael Ball

# Computational Structures in Data Science

---



## Lecture: Mutable Data

## Computing In the News [MIT News. Adam Zewe Jan 27, 2023]



### When Should Data Scientists Try a New Technique?

Researchers at the Massachusetts Institute of Technology, Columbia University, and the University of Wisconsin-Madison have formulated the "c-value" measure to help scientists choose which estimation methods to apply when modeling specific data problems. C-value helps users decide between approaches based on the odds that a new method will yield greater accuracy for a particular dataset. The c-value equation employs the dataset to first compute the estimate with each technique, then to compute the c-value between techniques. Methods with a high c-value are unlikely to be less accurate than the original method on the data problem. The researchers assessed c-value using real-world data problems, including modeling ocean currents, calculating violent crime in neighborhoods, and approximating students' reading skills in school





## Announcements

---

- Reminder: No Class Monday 2/20
- New Attendance Tracking Next Week
  - Not for points!!
  - We want you to attend in person
  - Will help us improve future offerings of 88C



UC Berkeley EECS  
Lecturer  
Michael Ball

# Computational Structures in Data Science

---



## Mutability



## Learning Objectives

---

- Distinguish between when a function mutates data, or returns a new object
  - Many Python "default" functions return new objects
- Understand modifying objects in place
- Python provides "is" and "==" for checking if items are the same, in different ways



## Why does Mutability Matter?

---

- Mutable data is a reality — lists, dictionaries, objects (coming soon)
- It's a challenging aspect of programming
- There are common patterns, which you will *slowly* become familiar with and internalize.
- **Use your environment diagrams!**



# Objects

---

- An **object** is a bundle of data and behavior.
- A type of object is called a **class**.
- Every value in Python is an object.
  - string, list, int, tuple, et
- All objects have attributes
- Objects often have associated methods
  - `lst.append()`, `lst.extend()`, etc
- **Objects have a value (or values)**
  - Mutable: We can change the object after it has been created
  - Immutable: We cannot change the object.
- Objects have an *identity*, a reference to that object.



## Immutable Object: string

---

- `course = 'CS88'`
- **What kind of object is it?**
  - `type(course)`
- **What data is inside it?**
  - `course[0]`
  - `course[2:]`
- **What methods can we call?**
  - `course.upper()`
  - `course.lower()`
- None of these methods modify our original string.





## Dictionaries are Mutable, too

---

- Immutable – the value of the object cannot be changed
  - integers, floats, booleans
  - strings, tuples
- Mutable – the value of the object can change
  - Lists
  - Dictionaries

```
>>> alist = [1,2,3,4]
>>> alist
[1, 2, 3, 4]
>>> alist[2]
3
>>> alist[2] = 'elephant'
>>> alist
[1, 2, 'elephant', 4]
```

```
>>> adict = {'a':1, 'b':2}
>>> adict
{'b': 2, 'a': 1}
>>> adict['b']
2
>>> adict['b'] = 42
>>> adict['c'] = 'elephant'
>>> adict
{'b': 42, 'c': 'elephant', 'a': 1}
```



## Dictionaries – by example

---

### Constructors:

```
dict( hi=32, lo=17)
dict([('hi',212),('lo',32),(17,3)])
{'x':1, 'y':2, 3:4}
{wd:len(wd) for wd in "The quick brown fox".split()}
```

### Selectors:

```
water['lo']
<dict>.keys(), .items(), .values()
<dict>.get(key [, default] )
```

### Operations:

```
in, not in, len, min, max
'lo' in water
```

### Mutators

```
water['lo' ] = 33
```



## Immutability vs Mutability

---

- An immutable value is unchanging once created.
- Immutable types (that we've covered): int, float, string, tuple

```
a_string = "Hi y'all"  
a_string[1] = "I" # ERROR  
a_string += ", how you doing?"  
an_int = 20  
an_int += 2
```

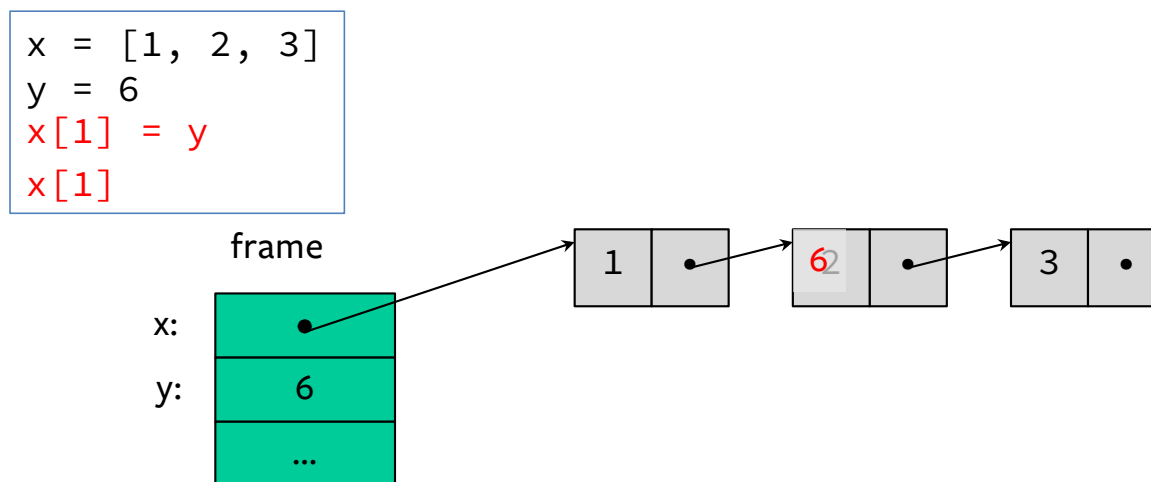
- A mutable value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.
- Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]  
grades_copy = grades  
grades[1] = 100 # grades_copy changes too!  
words = {"agua": "water"}  
words["pavo"] = "turkey"
```



## From value to storage ...

- A variable assigned a compound value (object) is a reference to that object.
- Mutable objects can be changed but the variable(s) still refer to it
  - x is still the same object, but it's values have changed.





## Mutating Lists: Example functions of the `list` class

---

- `append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

[Try in PythonTutor.](#)

- `extend()` adds all the elements in one list to another list:

```
s = [2, 3]
t = [5, 6]
s.extend(4) # ❌ Error: 4 is not an iterable!
s.extend(t)
t = 0
```

[Try in PythonTutor.](#) (After deleting the bad line)



## Mutating Lists -- More Functions!

---

- `list += [x, y, z]` # just like `extend`.
  - [You need to be careful with this one!](#) It modifies the list.

- `pop()` removes and returns the last element:

```
s = [2, 3]
```

```
t = [5, 6]
```

```
t = s.pop()
```

[Try in PythonTutor.](#)

- `remove()` removes the first element equal to the argument:

```
s = [6, 2, 4, 8, 4]
```

```
s.remove(4)
```

[Try in PythonTutor.](#)



# Mutation makes sharing visible

Python 3.6

```
1 x = 2
2 y = 3
3 print(x+y)
4 x = 4
→ 5 print(x+y)
```

[Edit this code](#)

Print output (drag lower right corner to resize)

```
5
7
```

Frames      Objects

Global frame	
x	4
y	3

Python 3.6

```
1 x = [1, 2, 3]
2 y = x
3 print(y)
4 x[1] = 11
→ 5 print(y)
```

[Edit this code](#)

Print output (drag lower right corner to resize)

```
[1, 2, 3]
[1, 11, 3]
```

Frames      Objects

Global frame	
x	→
y	→

list			
0	1	2	
1	11	3	



## Mutables Inside Immutables

---

- Mutable objects can "live" inside immutable objects!
- An immutable sequence may still change if it contains a mutable value as an element.
- Be **very careful**, and probably don't do this!

```
t = (1, [2, 3])
```

```
t[1][0] = 99
```

```
t[1][1] = "Problems"
```

- [Try in PythonTutor](#)





## Copies, 'is' and '=='

---

```
>>> alist = [1, 2, 3, 4]
>>> alist == [1, 2, 3, 4] # Equal values?
True
>>> alist is [1, 2, 3, 4] # same object?
False
>>> blist = alist # assignment refers
>>> alist is blist # to same object
True
>>> blist = list(alist) # type constructors copy
>>> blist is alist
False
>>> blist = alist[ : ] # so does slicing
>>> blist is alist
False
>>> blist
[1, 2, 3, 4]
>>>
```



## Equality vs Identity

---

```
list1 = [1,2,3]
```

```
list2 = [1,2,3]
```

- **Equality:** `exp0 == exp1` evaluates to True if both `exp0` and `exp1` evaluate to objects containing equal values (Each object can define what `==` means)

```
list1 == list2 # True
```

- **Identity:** `exp0 is exp1` evaluates to True if both `exp0` and `exp1` evaluate to the same object
- Identical objects always have equal values.

```
list1 is list2 # False
```

- [Try in PythonTutor.](#)



## What is the meaning of 'is'?

---

- is in Python means two items have the exact same *identity*
- Thus, `a is b` implies `a == b`
- Why? Each object has a function `id()` which returns its "address"
  - We won't get into what this means, but it's essentially an internal "locator" for that data in memory.
  - Think of two houses which have the exact same floor plan, look the same, etc. They are "the same house" but each has a unique address. (And thus are different houses)
- Think this is tricky? cool? amazing?
- Take CS61C (Architecture) and CS164 (Programming Languages)



UC Berkeley EECS  
Lecturer  
Michael Ball

# Computational Structures in Data Science

---



## Passing Data Into Functions



## Learning Objectives

---

- Passing in a mutable object in a function in Python lets you modify that object
- Immutable objects don't change when passed in as an argument
- Making a new name doesn't affect the value outside the function
- Modifying mutable data **does** modify the values in the parent frame.



## Mutating Input Data

---

- Functions can mutate objects passed in as an argument
- Declaring a new variable with the same name as an argument only exists within the scope of our function
  - You can think of this as creating a new name, in the same way as redefining a variable.
  - This will **not** modify the data outside the function, even for mutable objects.
- **BUT**
  - We can still directly modify the object passed in...even though it was created in some other frame or environment.
  - We directly call methods on that object.
- [View Python Tutor](#)



## Understanding Python: What should we return?

---

- Why do some functions return None?
- Why do some functions return a value?

Functions that mutate an argument should return None!

**C88C / 61A / Data Science View: Avoid mutating data unless it's necessary!**

Mutations are useful, but can get confusing quickly. This is why we focus on *functional programming* - map, filter, reduce, list comprehensions, etc.



## Python Gotcha's: `a += b` and `a = a + b`

---

- Sometimes similar *looking* operations have very different results!
- Why?
- `=` always binds (or rebinds) a value to a name.
- [Python Tutor](#)

```
def add_data_to_thing(thing, data):  
    print(thing)  
    thing += data  
    print(thing)  
    return thing
```

```
def new_thing_with_data(thing, data):  
    print(thing)  
    thing = thing + data  
    print(thing)  
    return thing
```





UC Berkeley EECS  
Lecturer  
Michael Ball

# Computational Structures in Data Science

---



## Mutable Functions



## Learning Objectives

---

- Remember: Each function gets its own new frame
- Inner functions can access data in the parent environment
- Use an inner function along with a mutable data type to capture changes



## Making Functions that Capture and change state

---

- We want to make a function, which returns a function that can change the state.
- [Python Tutor Link](#)

```
def make_counter():  
    counter = [0]  
    def count_up():  
        counter[0] += 1  
        return counter  
    return count_up
```

```
c = make_counter()  
print(c)  
c()  
c()  
c()
```



## Functions with Changing State

---

- Goal: Use a function to repeatedly withdraw from a bank account that starts with \$100.
- Build our account: `withdraw = make_withdraw_account(100)`
- First call to the function:  
`withdraw(25) # 75`
- Second call to the function:  
`withdraw(25) # 50`
- Third call to the function:  
`withdraw(60) # 'Insufficient funds'`



## How Do We Implement Bank Accounts?

---

- A mutable value in the parent frame can maintain the local state for a function.

```
def make_withdraw_account(initial):  
    balance = [initial]  
  
    def withdraw(amount):  
        if balance[0] - amount < 0:  
            return 'Insufficient funds'  
        balance[0] -= amount  
        return balance[0]  
    return withdraw
```

[View in PythonTutor](#)



## Implementing Bank Accounts

---

- A mutable value in the parent frame can maintain the local state for a function.

```
def make_withdraw_account(initial):  
    balance = [initial]  
  
    def withdraw(amount):  
        if balance[0] - amount < 0:  
            return 'Insufficient funds'  
        balance[0] -= amount  
        return balance[0]  
    return withdraw
```

[View in PythonTutor](#)



## OPTIONAL: Modifying Parent frames. (global and nonlocal)

```
>>> counter = 0
>>> def count_fun():
...     global counter
...     counter += 1
...     return counter
...
>>> count_fun()
1
>>> count_fun()
2
```

How do I make a second counter?

```
>>> def make_counter():
...     counter = 0
...     def counts():
...         nonlocal counter
...         counter += 1
...         return counter
...     return counts
...
>>> count_fun = make_counter()
>>> count_fun()
1
>>> count_fun()
2
>>> another_one = make_counter()
>>> another_one()
1
>>> count_fun()
3
```