



UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science



Lecture 14:

Mutability

Announcements



Computing In the News





UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science



Mutability: Lists



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



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
- **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_tuple = (1, 2)
a_tuple[0] = 3 # AN ERROR
a_string = "Hi y'all"
a_string[1] = "I" # ERROR
a_string += ", how you doing?"
an_int = 20
an_int += 2
```

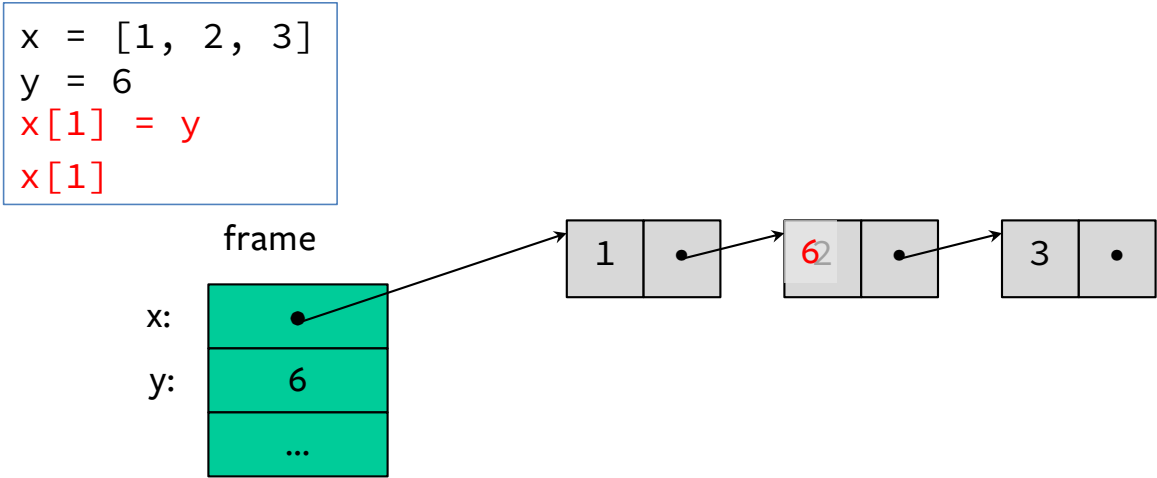
- A mutable value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.
- Mutable types (that we've covered): list, dict

```
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

- `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 a 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

- Mutables inside immutables
- An immutable sequence may still change if it contains a mutable value as an element.

```
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
```

- [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`
- 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 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](#)



Example

```
def add_data_to_obj(obj, data):  
    print(obj)  
    obj += data  
    print(obj)  
    return obj
```

```
def new_obj_with_data(obj, data):  
    print(obj)  
    obj = obj + data  
    print(obj)  
    return obj
```



UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science



Mutable Functions



Learning Objectives

- A function has its own environment
- Inner functions can access data in the parent environment
- Use an inner function along with a mutable data type to capture changes



Functions with Changing State

- Goal: Use a function to repeatedly withdraw from a bank account that starts with \$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'`



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](#)