UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Lecture:
Mutable Data

Announcements

- Midterm Scores out on the Weekend / Monday
- Ants project coming out soon.
  - Puts OOP into practice!
- Next few weeks, some big ideas in CS!
  - Today: Solidify some understandings of data structures
  - Next up: Efficiency
  - Soon: Linked-Lists and Trees (great 61B prep!)
- End: SQL. Foundational for Data Science

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

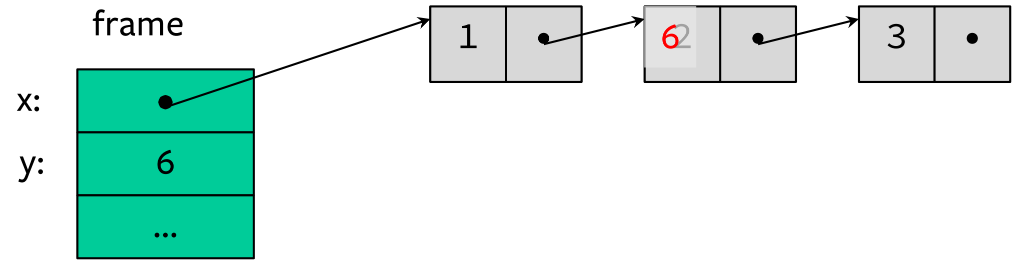
- A mutable value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.
- Mutable types (that we've covered): list, dict

```
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.

```
x = [1, 2, 3]
y = 6
x[1] = y
x[1]
```



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 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()
```

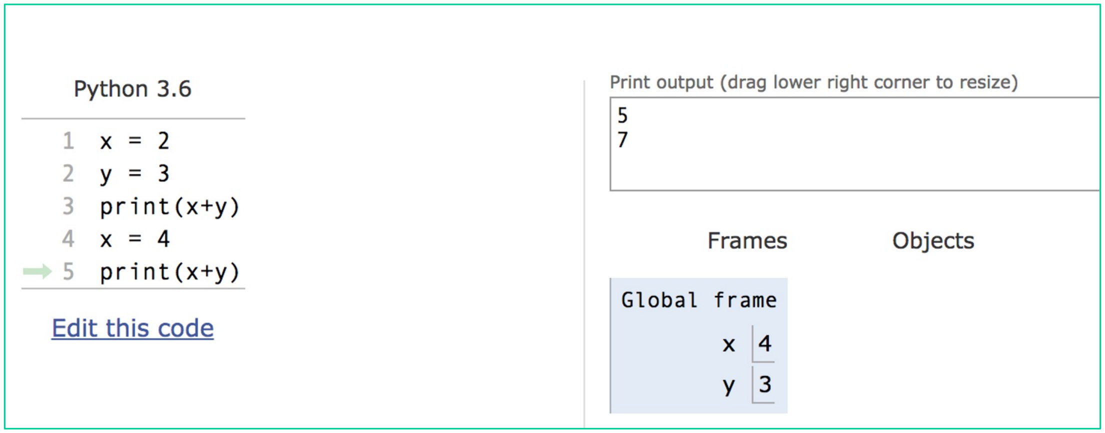
Try in PythonTutor.

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

```
s = [6, 2, 4, 8, 4]
s.remove(4)
```

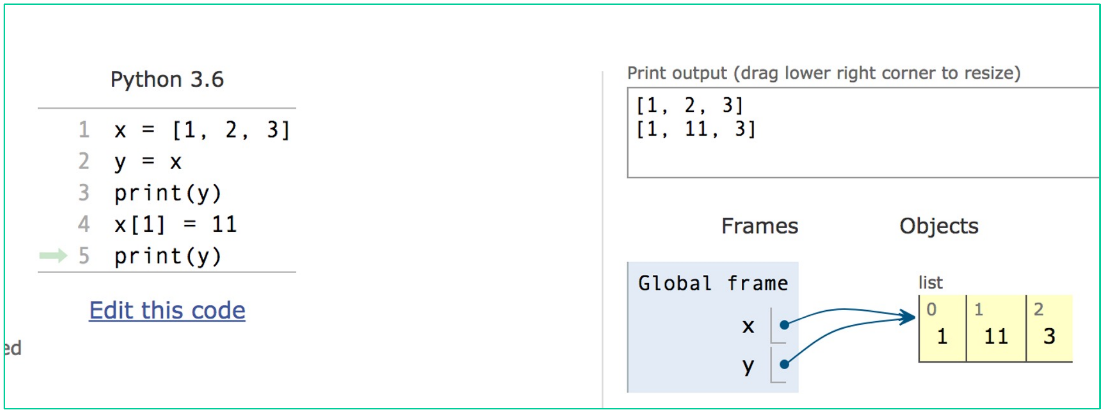
Try in PythonTutor.

Mutation makes sharing visible



```
x = 2
y = 3
print(x+y)
x = 4
print(x+y)
```

```
5
7
```



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

```
[1, 2, 3]
[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`
- 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

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.
- += maps to the special method, e.g. **`__iadd__`**

```
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

- 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

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

global and nonlocal allow us to modify parent frames.

```
>>> 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
```