Computational Structures in Data Science

Lecture:
Mutable Data

Week 3, Summer 2024. 7/2 (Tues)

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- HW04, Lab04 released today (Due: 7/8) (+2 days due to holiday)
- Project01 (“Maps”) will be released on Wednesday
  - Partner project (though you can solo it if you’d prefer)
- Reminder: no lecture/labs/OH on Thursday (July 4th)

Lecture overview

- Mutability
- Identity (“is” vs “==“)
- Mutable functions (aka “functions with state”)

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.

Python Tutor: Assignments Are References

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

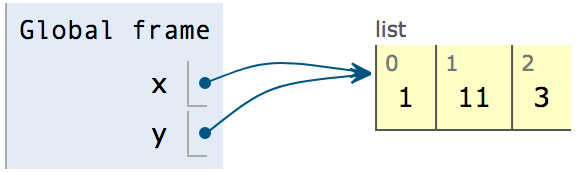
```
1  x = [1, 2, 3]
2  y = x
3  print(y)
4  x[1] = 11
5  print(y)
```

Edit this code

Print output (drag lower right corner to resize)

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

Frames Objects



Mutable Data Inside Immutable Objects

- 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 **do not** do this!

```
t = (1, [2, 3])
t[1][0] = 99
t[1][1] = "Problems"
```

- Try in PythonTutor

Equality vs Identity

- We know how to use "==" comparison operator to see if two things have the same **value**

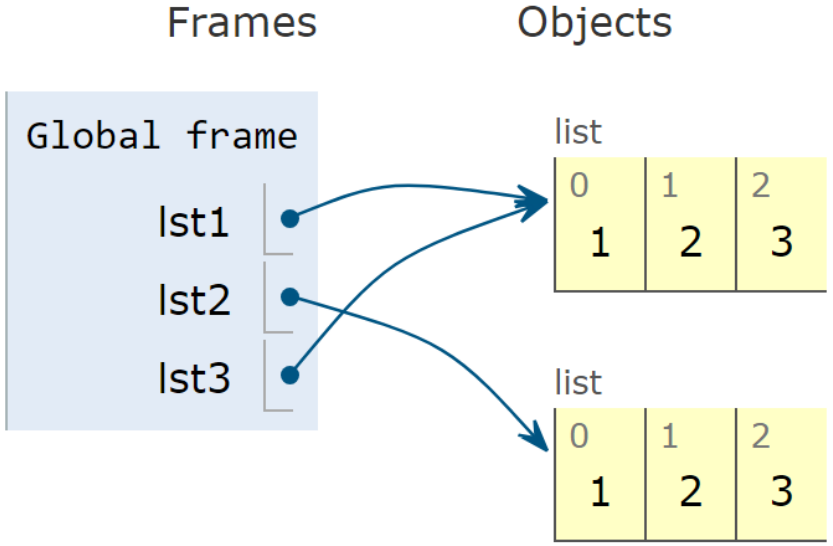
```
>>> my_name = "eric"
>>> my_name == "eric" # True
```

- With **mutable objects**, it's now important to keep track of which **object** a variable is **referencing**

Python 3.6
(known limitations)

```
1  lst1 = [1, 2, 3]
2  lst2 = [1, 2, 3]
3  lst3 = lst1
```

Edit this code

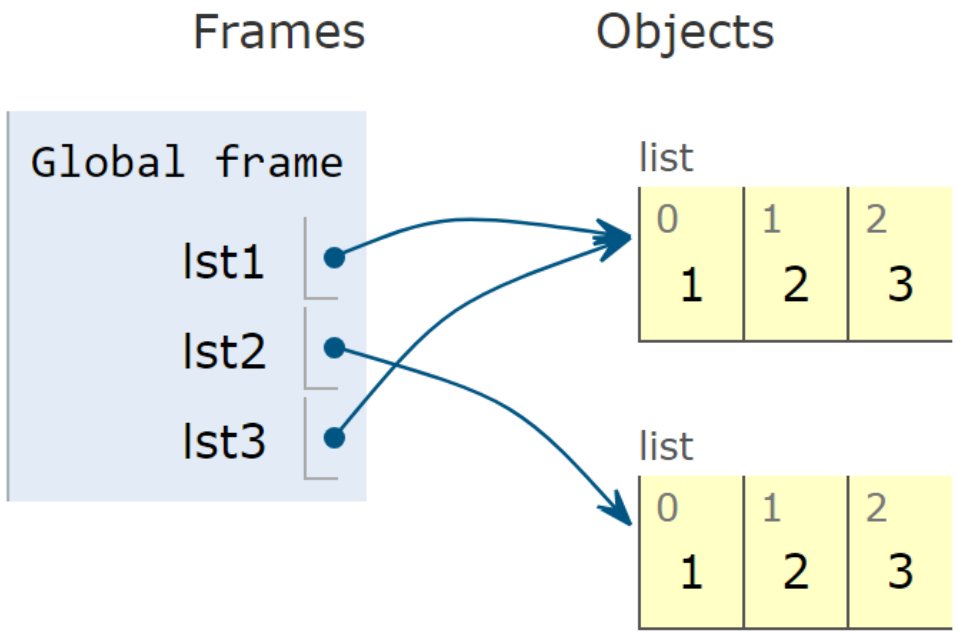


Equality vs Identity

Python 3.6
(known limitations)

```
1  lst1 = [1, 2, 3]
2  lst2 = [1, 2, 3]
3  lst3 = lst1
```

Edit this code



- lst1 and lst2 point to different objects, but have the same values

=> Equality (lst1 == lst2)

Aka “Value equality”

- lst1 and lst3 point to the same object (and, as a result, have the same values)

=> Identity (`lst1 is lst3`)

Aka “Identity equality”

Identity and == vs is

How do we know if two names (variables) are the same exact object? i.e. Will modifying one modify the other?

```
>>> 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]
>>>
```

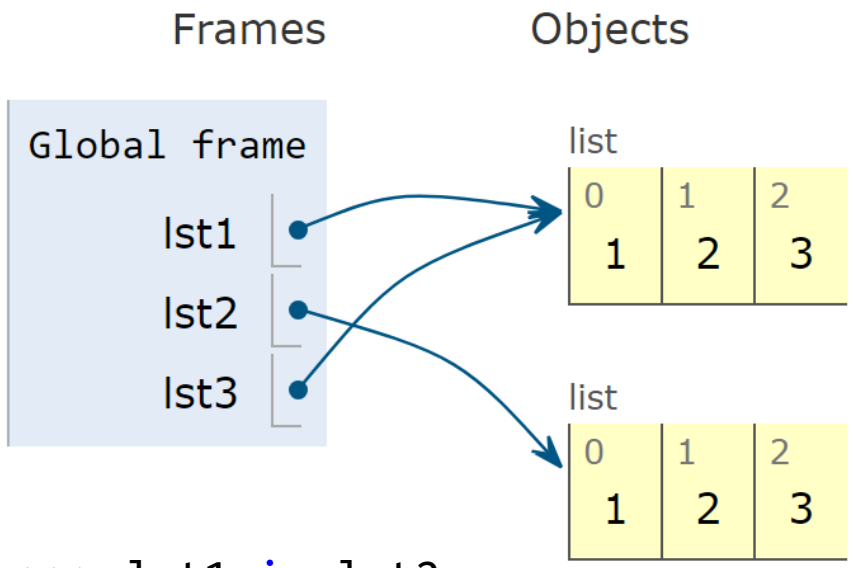
What is the meaning of `is`?

- How does Python implement the `is` operator?
- **Intuitive (visual) answer**: `a is b` is True iff `a` **points** to the same exact object as `b` points to.

Python 3.6
(known limitations)

```
1  lst1 = [1, 2, 3]
2  lst2 = [1, 2, 3]
3  lst3 = lst1
```

Edit this code




```
>>> lst1 is lst2
False
>>> lst1 is lst3
True
```

What is the meaning of `is`?

- How does Python implement the `is` operator?
- **Technical answer**: `a is b` is True iff `a` points to the same **memory address** as `b` points to.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst3 = lst1
>>> id(lst1)
1906951142144
>>> id(lst2)
1906951128448
>>> id(lst3)
1906951142144
```

These are memory addresses! Eg somewhere in your CPU RAM

Tip: think of RAM as a long array/list, and an address as an index into the list.

```
>>> lst1 is lst2
False
>>> lst1 is lst3
True
>>> id(lst1) == id(lst2)
False
>>> id(lst1) == id(lst3)
True
```

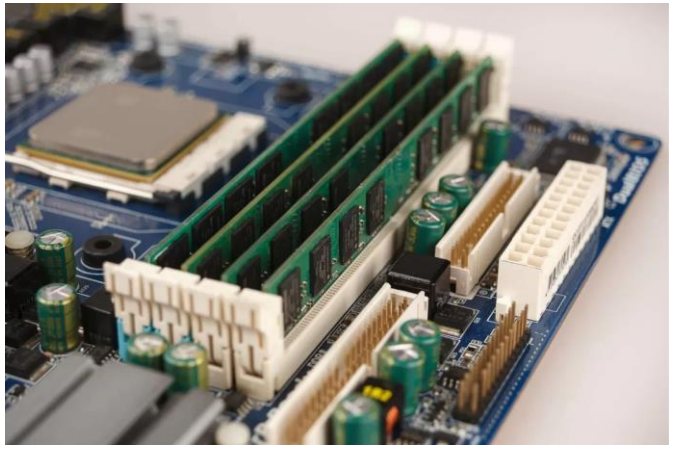
Thus, the `is` operator is actually comparing memory addresses behind the scenes.

Python's `id()` function

- In Python, each object has a function `id()` which returns its "**memory address**"
- CPU memory (random access memory, aka "RAM")
- Example: In 2024, a Macbook Pro 14" has 8 GB - 36 GB CPU RAM
  - Aside: GPU (graphical processing units) have their own separate GPU memory. State-of-the-art ML models (like ChatGPT, etc) are notoriously GPU-memory intensive

- Bill Gates once said* in 1981 "**640K of memory should be enough for anybody.**"
  - * Not actually true, but it makes a funny story

- Think this is cool? amazing?
- Related courses: CS61C (Architecture), CS162 (Operating Systems), CS164 (Programming Languages and Compilers)



Credit: https://www.techspot.com/article/2024-anatomy-ram/

Computational Structures in Data Science

Passing Data Into Functions

Berkeley
UNIVERSITY OF CALIFORNIA

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 Arguments

- 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?

Convention: Functions that mutate an argument **usually** 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.

Functions that Mutate vs Return New Objects

- Lists:
  - sorted(list) – returns a new list
  - list.sort() – modifies the list, **returns None**
  - list.append() – modifies the list, **returns None**
  - list.extend() – modifies the list, **returns None**

The fact that these return None are a strong hint that these functions mutate the object (“in place” modification”)

How to be sure? Read the function documentation.

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

- Sometimes similar ***looking*** operations have very different results!
- Hint: = always binds (or re-binds) a value to a name.
- Python Tutor

```
def add_data_to_thing(thing, data):
    print(f"+=, Before: {thing}")
    thing += data
    print(f"+=, After: {thing}")
    return thing

def new_thing_with_data(thing, data):
    print(f"=, Before: {thing}")
    thing = thing + data
    print(f"=, After: {thing}")
    return thing
```

Main takeaway:
For lists, the `lst += thing` operator is NOT the same as `lst = lst + thing`.

Instead, list's `+=` operator is an in-place modification of the LHS.

Tip: `lst += thing` calls the list.__iadd__() function, which performs the in-place modification.

Computational Structures in Data Science

Mutable Functions

Berkeley
UNIVERSITY OF CALIFORNIA

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

Putting it all together: a Counter

- Applying everything we've learned, we're now ready to implement a Counter function
- It will "remember" its state, and each time we call it, will increment its internal counter value

```
# Desired usage
>>> counter = make_counter()
>>> counter()
1
>>> counter()
2
>>> counter()
3
```

```
def make_counter():
    # IMPLEMENT ME
    pass
```

Question: how to finish the `make_counter()` definition?

Putting it all together: a Counter

- Applying everything we've learned, we're now ready to implement a Counter function
- It will "remember" its state, and each time we call it, will increment its internal counter value

```
# Desired usage
>>> counter = make_counter()
>>> counter()
1
>>> counter()
2
>>> counter()
3
```

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

Python Tutor link

Tip: step through the Python Tutor visualization to see what's going on!

Putting it all together: a Counter

- Here is an alternate implementation. What Would Python Print?

```
def make_counter():
    my_state = [0]
    def count_up():
        my_state.append(my_state[-1] + 1)
        return my_state[-1]
    return count_up
```

```
>>> counter = make_counter()
>>> counter()
# FILL ME IN
>>> counter()
# FILL ME IN
>>> counter()
# FILL ME IN
```

Putting it all together: a Counter

- Here is an alternate implementation. What Would Python Print?

```
def make_counter():
    my_state = [0]
    def count_up():
        my_state.append(my_state[-1] + 1)
        return my_state[-1]
    return count_up
```

```
>>> counter = make_counter()
>>> counter()
1
>>> counter()
2
>>> counter()
3
```

It works! Hooray.

Question: what is one downside to this implementation?

Putting it all together: a Counter

- Here is an alternate implementation. What Would Python Print?

```
def make_counter_v2():
    my_state = [0]
    def count_up():
        my_state.append(my_state[-1] + 1)
        return my_state[-1]
    return count_up
```

```
>>> counter = make_counter_v2()
>>> counter()
1
>>> counter()
2
>>> counter()
3
```

It works! Hooray.

Question: what is one downside to this implementation?
Answer: the `my_state` list will grow for each call. If we call it many times, the memory usage may grow too high and crash the machine (“CPU out of memory”).

Putting it all together: a Counter

- With the same implementation, What Would Python Print?

```
def make_counter_v2():
    my_state = [0]
    def count_up():
        my_state.append(my_state[-1] + 1)
        return my_state[-1]
    return count_up
```

```
>>> make_counter_v2()()
# FILL ME IN
>>> make_counter_v2()()
# FILL ME IN
```

Putting it all together: a Counter

- With the same implementation, What Would Python Print?

```
def make_counter_v2():
    my_state = [0]
    def count_up():
        my_state.append(my_state[-1] + 1)
        return my_state[-1]
    return count_up
```

```
>>> make_counter_v2()()
1
>>> make_counter_v2()()
1
```

It doesn't work! Gasp.

Root cause: each call to `make_counter_v2()` creates a NEW `count_up()` function with a fresh `my_state = [0]`.

Not following? Try following the Python Tutor visualization.

Functions with state: Bank account

- 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.
- View in PythonTutor

```
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
```

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

Aside: Counters and nonlocal/global

- Here is another implementation. What Would Python Print?

```
def make_counter():
    my_state = 0
    def count_up():
        my_state = my_state + 1
        return my_state
    return count_up
```

```
>>> counter = make_counter()
>>> counter()
# FILL ME IN
>>> counter()
# FILL ME IN
>>> counter()
# FILL ME IN
```

Aside: Counters and nonlocal/global

- Here is another implementation. What Would Python Print?

```
def make_counter():
    my_state = 0
    def count_up():
        my_state = my_state + 1
        return my_state
    return count_up
```

```
>>> counter = make_counter()
>>> counter()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in count_up
UnboundLocalError: local variable
'my_state' referenced before assignment
```

Woah. What is going on here?

Due to language design decisions, Python does not let you re-bind variables that exist in parent frames (including the global frame).

For a more detailed explanation straight from the Python3 docs, see this link.

Aside: Counters and nonlocal/global

- One way to fix this is to use the nonlocal keyword

```
def make_counter_v3():
    my_state = 0
    def count_up():
        nonlocal my_state
        my_state = my_state + 1
        return my_state
    return count_up
```

```
>>> counter = make_counter()
>>> counter()
1
>>> counter()
2
```

`nonlocal my_state` tells Python that the `my_state` variable refers to a variable in a parent (non-local) frame, and any assignments should modify the variable in the OTHER frame.

Aside: Counters and nonlocal/global

- There's an analogous keyword for global vars, `global`

```
a_global_var = 42
def fn():
    global a_global_var
    a_global_var = 9000

>>> print(f"Before fn(): {a_global_var}")
Before fn(): 42
>>> fn()
>>> print(f"After fn(): {a_global_var}")
After fn(): 9000
```

In Data C88C su24: nonlocal, global

- In this class (Data C88C Summer 2024), we will NOT be using the Python keywords: `nonlocal`, `global`
- For assignments/exams, we won't expect you to use nonlocal/global
- But, we will expect you to understand why this doesn't work

```
def make_counter():
    my_state = 0
    def count_up():
        my_state = my_state + 1
        return my_state
    return count_up
```

```
>>> counter = make_counter()
>>> counter()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in count_up
UnboundLocalError: local variable
'my_state' referenced before assignment
```

Aside: Counters and nonlocal/global

- (review) What Would Python Print?

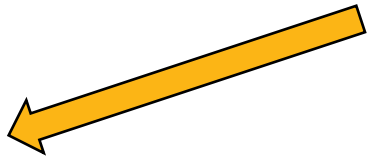
```
a_global_var = 42
def fn():
    a_global_var = 9000

>>> print(f"Before fn(): {a_global_var}")
42
>>> fn()
>>> print(f"After fn(): {a_global_var}")
# FILL ME IN
```

Aside: Counters and nonlocal/global

- (review) What Would Python Print?

```
a_global_var = 42
def fn2():
    a_global_var = 9000
```



This creates a new variable `a_global_var` in the fn2 frame, with value 9000. Notably, this `a_global_var` shadows the `a_global_var` at the global frame.
Visualization: Python Tutor

```
>>> print(f"Before fn(): {a_global_var}")
42
>>> fn2()
>>> print(f"After fn(): {a_global_var}")
42
```

Lecture overview. Any questions?

- Mutability
- Identity (“is” vs “==“)
- Mutable functions (aka “functions with state”)

