Computational Structures in Data Science

Environments, Mutable Data

Week 3, Summer 2024. 7/1 (Thurs)

Berkeley
UNIVERSITY OF CALIFORNIA

Photo by Stephanie McCabe on Unsplash

Announcements

- Upcoming due dates
  - HW03, Lab03: Due 7/1 11:59 pm PST (tonight!)
- Project01 (“Maps”) will be released later this week
  - You can work on this + submit with one partner.
    - Or, you can solo it.
- (Holiday) July 4: No lecture/labs/OH!

Credit: Stephanie McCabe: https://unsplash.com/photos/time-lapse-photography-of-sparkler-and-usa-flag-let-_Ajm-ewEC24

Lecture overview

- Environment diagrams (review, lambdas)
- Mutable data
  - dict

Computational Structures in Data Science

Environment Diagrams (review, lambdas)

Berkeley
UNIVERSITY OF CALIFORNIA

Inner (“Nested”) Functions

- Inner functions are *scoped* – they are not visible to the outside world
- But they can be *returned* and thus called later on.
- Like a "regular" function, they have access to all the data (including arguments) of their "parent" or "container" function.

```
def outer_fn(arg_outer):
    def inner_fn(arg_inner):
        return arg_outer + arg_inner
    return inner_fn
```

```
>>> outer_fn(2)(3)
5
```


Learning Objectives

- Lambda are anonymous functions, which are expressions
  - Don't use `return`, lambdas always return the value of the expression.
  - They are typically short and concise
  - They don't have an "intrinsic" name when using an environment diagram.
    - Their name is the character λ

(review) lambda syntax

Function expression

"anonymous" function creation

lambda <arg or arg_tuple> : <expression using args>

Expression, not a statement, no return or any other statement

```
add_one = lambda v : v + 1
```

```
def add_one(v):
    return v + 1
```


Examples

```
>>> def make_adder(i):
...     return lambda x: x+i
...
>>> make_adder(3)
<function make_adder.<locals>.<lambda> at
0x10073c510>

>>> make_adder(3)(4)
7

>>> list(map(make_adder(3), [1,2,3,4]))
[4, 5, 6, 7]
```

Environment Diagrams (review)

- Organizational tools that help you understand code
- Terminology:
  - **Frame:** keeps track of variable-to-value bindings, each function call has a frame
  - **Global Frame:** global for short, the starting frame of all python programs, doesn't correspond to a specific function
  - **Parent Frame:** The frame of where a function is defined (default parent frame is global)
  - **Frame number:** What we use to keep track of frames, f1, f2, f3, etc
  - **Variable** vs **Value**: x = 1. x is the **variable**, 1 is the **value**

Environment Diagrams Rules (review)

1. Always draw the global frame first
2. When evaluating assignments (lines with single equal), always evaluate right side first
3. When you CALL a function MAKE A NEW FRAME!
4. When assigning a primitive expression (number, boolean, string) write the value in the box
5. When assigning anything else (lists, functions, etc.), draw an arrow to the value
6. When calling a function, name the frame with the intrinsic name – the name of the function that variable points to
7. The parent frame of a function is the frame in which it was defined in (default parent frame is global)
8. If the value for a variable doesn't exist in the current frame, search in the parent frame

(redo from lec07) Python Tutor Example #3 (LINK)

```
def make_adder(a):
    return lambda x: x + a

add_2 = make_adder(2)
add_3 = make_adder(3)

x = add_2(2)

def compose(f, g):
    def h(x):
        return f(g(x))
    return h

add_5 = compose(add_2, add_3)
z = add_5(x)
```

Main takeaway from this:

- A function's parent frame is the frame in which it was created, NOT where it is called from!
  - Aka "lexical/static" scoping.
- Lambdas behave just like functions, but with "no name" (anonymous fn)
- Variable "shadowing": the `x` in `h()` frame "**shadows**" the `x` in the global frame

Computational Structures in Data Science

Sorting + HOF's

Berkeley
UNIVERSITY OF CALIFORNIA

More Python HOFs

- sorted – sorts a list of data
- min
- max

All three take in an optional argument called **key** which allows us to control how the function performs its action. They are more similar to filter than map.

```
max([1,2,3,4,5], key = lambda x: -x)
```

key is the name of the argument and a lambda is its value.

```
>>> fruits = ["pear", "grape", "KIWI", "APPLE", "melon",
"ORANGE", "BANANA"]
>>> sorted(fruits key=lambda x: x.islower())
['KIWI', 'APPLE', 'ORANGE', 'BANANA', 'pear', 'grape',
'melon']
```


Sorting Data

- It is often useful to sort data.
- What property should we sort on?
  - Numbers: We can clearly sort.
  - What about the length of a word?
  - Alphabetically?
  - What about sorting a complex data set, but 1 attribute?
    - Image I have a list of courses: I could sort be course name, number of units, start time, etc.
- Python provides 1 function which allows us to provide a *lambda* to control its behavior

Sorting with Lambdas

```
>>> sorted([1,2,3,4,5], key = lambda x: x)
[1, 2, 3, 4, 5]
>>> sorted([1,2,3,4,5], key = lambda x: -x)
[5, 4, 3, 2, 1]
# Sorting a list of tuples by various criterion
>>> my_tuples = [(2, "hi"), (1, "how"), (5, "goes"), (7, "it")]
# Sort by first entry (int)
>>> sorted(my_tuples, key = lambda x: x[0])
[(1, 'how'), (2, 'hi'), (5, 'goes'), (7, 'it')]
# Sort by second entry (str)
>>> sorted(my_tuples, key = lambda x: x[1])
[(5, 'goes'), (2, 'hi'), (1, 'how'), (7, 'it')]
# Sort by length of second entry
>>> sorted(my_tuples, key = lambda x: len(x[1]))
[(2, 'hi'), (7, 'it'), (1, 'how'), (5, 'goes')]
```

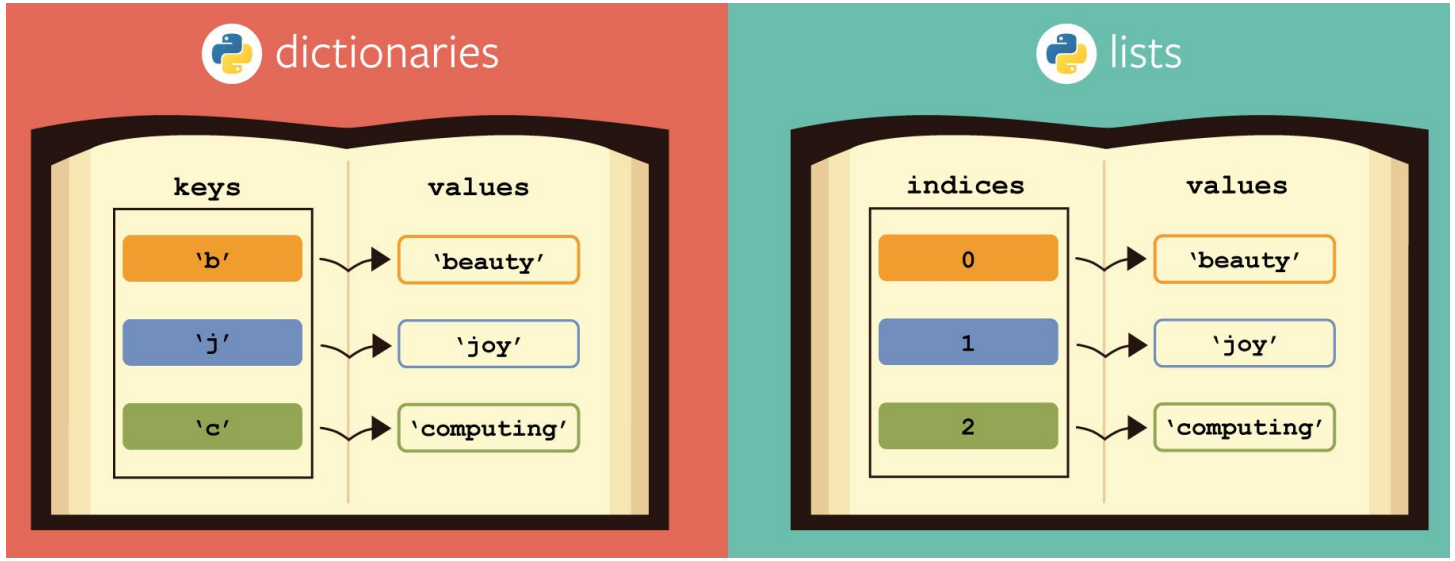

Computational Structures in Data Science

Dictionaries

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Dictionaries are a new type in Python
- **Lists** let us index a value by a number, or position.
- **Dictionaries** let us index data by other kinds of data.
  - KEY -> VALUE mapping. Aka "**lookup table**"
  - Extremely useful data type used in many languages/systems



Credit: https://berkeley-cs61as.github.io/textbook/dictionaries-and-memoization.html

Dictionaries: syntax

```
# Constructors
# dict( <list of 2-tuples> )
my_dict = dict(('key1', 'value1'), ('key2', 'value2'))
# dict( <key>=<val>, ...) # like kwargs
dict(key1='value1', key2='value2')
# { <key exp>:<val exp>, ... }
{"key1": "value1", "key2": "value2"}

# { <key>:<val> for <iteration expression> }
{key: value for (key, value) in [("key1", "value"), ("key2", "value2")]}
# Example:
>>> {x: y for x, y in zip(["a", "b"], [1, 2])}
{'a': 1, 'b': 2}
```

Dictionaries: common operations

```
my_dict = {"key1": "value1", "key2": "value2"}

# selector (key present / missing)
>>> my_dict['key1']
'value1’
>>> my_dict['meow']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'meow’
>>> my_dict.get('meow')  # returns None
>> my_dict.get('meow', 'fallback')
'fallback'
```

```
# operators: in, not in, min/max
>>> 'key1' in my_dict
True
>>> 'not_present' in my_dict
False


# writing to a dict
my_dict['key3'] = 42
my_dict['key3']
42
```


Demo

```
person = { 'name': 'Michael' }
person.get('name')
person['email'] = 'ball@berkeley.edu'
person.keys()
'phone' in person


text = 'One upon a time'
{ word : len(word) for word in text.split() }
```


Computational Structures in Data Science

Mutability

Berkeley
UNIVERSITY OF CALIFORNIA

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?

- Recall: "**mutable/mutation**" means "to modify / modifiable"
  - "**immutable**" means "unmodifiable"
- 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 in Python (preview of Object-Oriented Programming)

- 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.

Mutable Objects: lists and dictionaries

- **Immutable** – the value of the object cannot be changed
  - integers, floats, booleans
  - strings, tuples
- **Mutable** – the value of the object can change over time
  - 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 (syntax/operations review)

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
'name' in course
```

Mutators

```
course['number' ] = 'C88C'
course.pop('room')
del course['room']
```

These mutator operations modify the dict object!

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
```

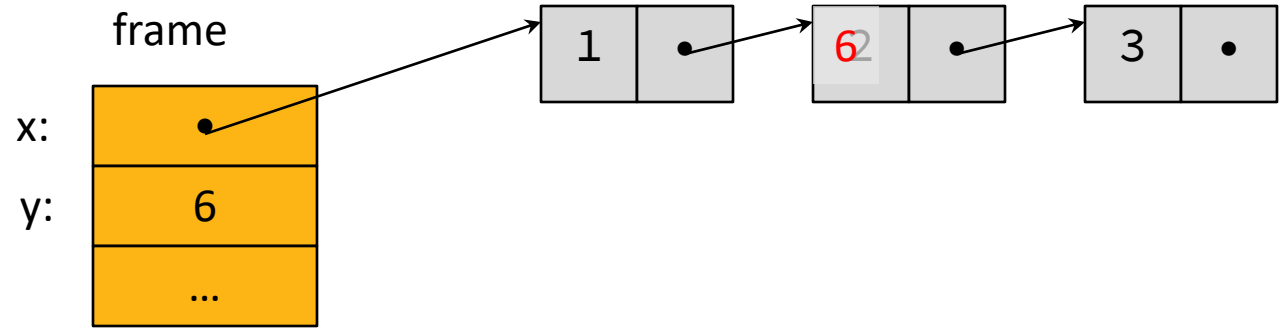
- A mutable value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.
- Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]
grades_copy = grades # Not actually a copy!
grades[1] = 100 # grades_copy changes too!
words = {"agua": "water"}
words["pavo"] = "turkey"
```

Mutation in Environments

- 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 its 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 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

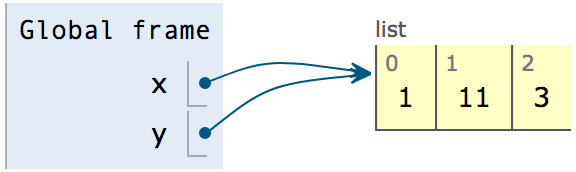
```
1  x = [1, 2, 3]
2  y = x
3  print(y)
4  x[1] = 11
5  print(y)
```

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

```
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.

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`?

- 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. The are "the same house" but each have a unique address. (And thus are different houses)

- Think this is tricky? cool? amazing?
- Take CS61C (Architecture) and CS164 (Programming Languages)

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?

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) – retiurns a new list
  - list.sort() – modifies the list, returns None
  - list.append() – modifies the list, returns None
  - list.extend() – modifies the list, returns None

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

- Sometimes similar ***looking*** operations have very different results!
- Why?
- = 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
```

Lecture overview. Any questions?

- Environment diagrams (review, lambdas)
- Mutable data
  - dict

