Discussion 6

Mutability, Iterators, Generators

Antonio Kam

anto [at] berkeley [dot] edu

Announcements

- There's like nothing to say lol
- Fill out the mid-semester feedback form! Super useful
  - All responses are anonymous, and I read everything that shows up. It'll help improve your experience in discussion
  - pls fill this out
  - be honest as well, i'll greatly appreciate it

Comments from last section

- Real world examples of trees in Python
  - Binary Search Trees (you'll see this in CS 61B)
  - Makes searching for ordered items very easy and fast
- Your take on AI being really good at mimicking peoples voices.
  - 🥶
- Best TV show/movie of all time?
  - HTTYD

Temperature Check 🌡️

- Mutability
- Iterators
- Generators (today's lecture, I think)

All slides can be found on

`teaching.rouxl.es`

Mutability

List Mutation Functions (adding)

- `.append(element)`
  - Adds elements to the end of the list
  - All elements go in one new box (can get nested lists if the element passed in is a list)
- `.extend(iterable)`
  - Concatenates two lists together (typcially `iterable` is a list)
- `.insert(index, element)`
  - Inserts `element` at `index`
  - Does **not** replace elements - this operation instead makes the list longer.
- All these functions return `None` once you use them

List Mutation Functions (removing)

- `.remove(element)`
  - Removes first appearance of element in list
  - Errors if it's unable to remove an element
- `.pop(optional index)`
  - Removes and **returns** element at the given index
  - If index is not provided, it defaults to the last element in the list.

Mutating Lists

- List mutation functions can modify an existing list
- Slicing will create a new list
  - Examples later
- `a = a + b` will create a new list
- `a += b` does not create a new list
- Indexing into a list and changing the element at that list will mutate the list:
  - `a[0] = 7` will change the first element in `a` to be 7.

Identity vs Equality

- `is` will check whether 2 objects are the same thing (i.e. pointing to the same object)
- `==` will check if two objects have the same value

Identity vs Equality

- `is` will check whether 2 objects are the same thing (i.e. pointing to the same object)
- `==` will check if two objects have the same value

```
a = [1, 2, 3]
b = [1, 2, 3]
a == b # True
a is b # False
```

Mutating Lists (Example)

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

test1a = lst1 == lst2
test1b = lst1 == lst3
test2a = lst1 is lst2
test2b = lst1 is lst3

lst1.append(3)
lst1 = lst1 + [4]
```

(For those reading the slides later, put this into tutor.cs61a.org)

Shallow Copy vs Deep Copy

- Shallow Copy
  - Only copies the first layer of a list
  - If we have a nested list, we only copy the arrow (not the list itself)
  - Create a new list where you copy over whatever is in the same box
- Deep Copy
  - Makes a complete copy of everything in a list
  - Very slow operation - no easy way to do this
- Python uses shallow copies (as do most languages) when copying lists!

Example: Shallow Copy vs Deep Copy

```
lst1 = [1, 2, [3, 4], 5]
lst2 = lst1[:]
```

(For those reading the slides later, put this into tutor.cs61a.org)

Worksheet!

Iterators

Iterators

- Iterable
  - Old View: Something we can go through one at a time
  - New View: Something we can call `iter` on - this will return an iterator.
- Iterator
  - A type of object that lets us keep track of which element is next in the sequence

Functions to Interact with Iterators

- `iter(iterable)`
  - Takes in an iterable and returns an iterator.
    - Calling `iter` on an iterable makes a brand new iterator
    - Calling `iter` on an iterator returns that same iterator (not a copy!) and does not change the state of that iterator
- `next(iterator)`
  - Takes in an iterator and outputs the next element
    - Raises `StopIteration` when it has no more elements to go through
    - Cannot call `next` on an *iterable*

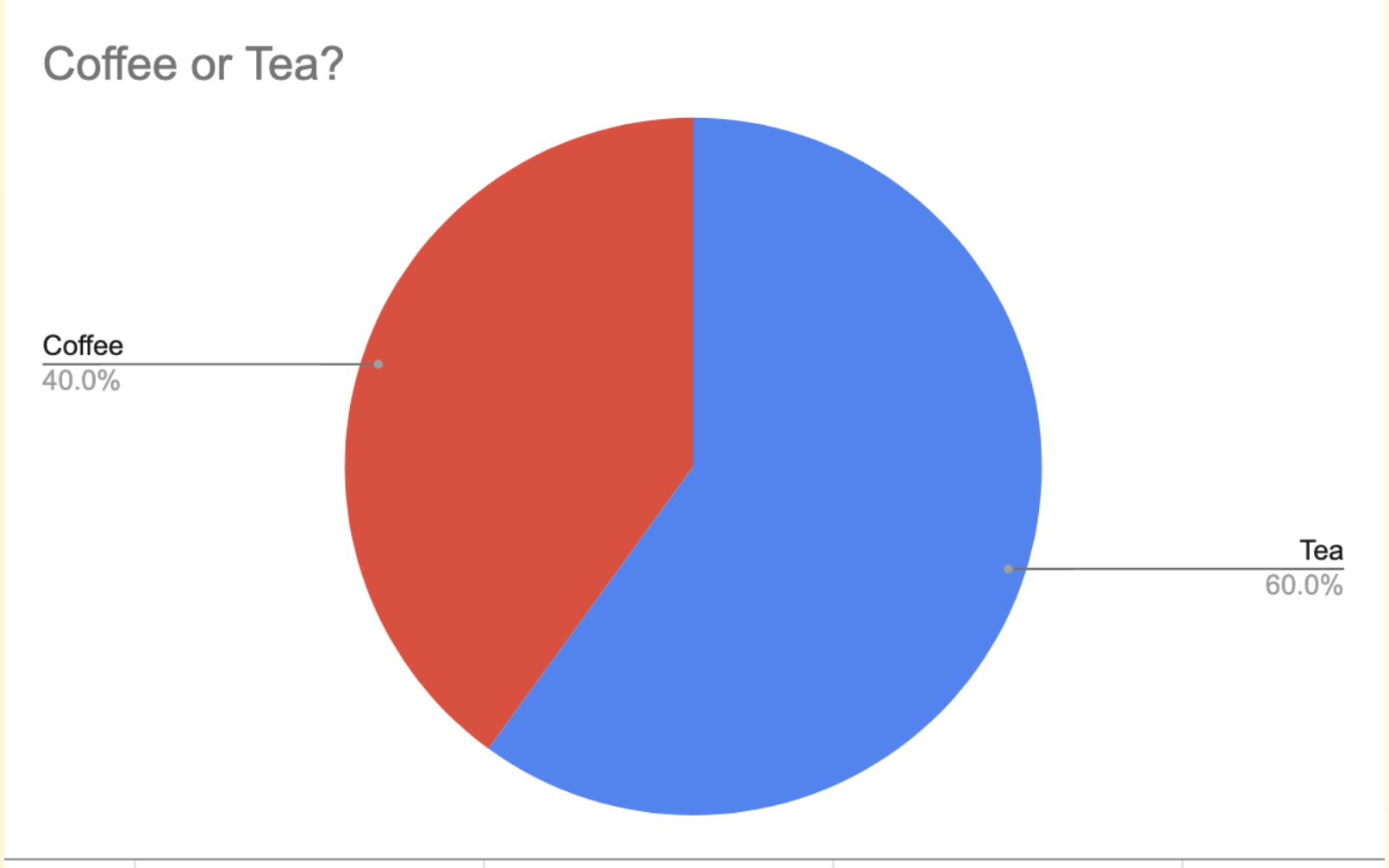
Analogy

- Iterable = Book
- Iterator = Bookmark
- Calling `iter` on an iterable (book) will create a new bookmark
- Calling `iter` on an iterator (bookmark) will just give you the same bookmark (no reason to mark where a bookmark is)
- Calling `next` on an iterator moves the bookmark to the next chapter

do some terminal example pls thx

Worksheet!

Results from last section (`links.rouxl.es/disc`)



Generators

Generators

- Generators are more specific versions of iterators that you can create yourself!
- When a generator function is called, it returns a generator object!
- The body of a generator function is not evaluated until `next` is called on the returned generator object.
- Generator functions look like normal functions, but use `yield` instead of `return`. Python will automatically see `yield` and determine a function a generator function if necessary.

Generators (continued)

- When `next` is called on a generator object, the body of a function is executed until `yield` is reached.
- From there, the yield statement will return a value, and then pauses that specific function at that moment (by saving the frame, and the line that it's on)
- When `next` is called again, we continue where we left off, until `yield` is reached again.
- `StopIteration` will be raised at the end of a generator function

Example

```
def countdown_generator(n):
    assert n >= 0
    while n >= 0:
        yield n
        n -= 1
```

`yield from`

- You can use `yield from <iterable>` if you want to yield every value from an iterable. These are equivalent:

```
lst = [1, 2, 3]
# Version 1
yield from lst

for item in lst:
    yield item
```

Mental Health Resources

- CAPS:
  - If you need to talk to a professional, please call CAPS at 510-642-9494.
- After Hours Assistance
  - For any assistance after hours, details on what to do can be found at this link

Anonymous Feedback Form

`links.rouxl.es/feedback`

Thanks for coming! 🥳

Please give me feedback on what to improve!