Let’s imagine you order a mushroom and cheese pizza from Domino’s, and that they represent your order as a list:

```python
>>> pizza1 = ['cheese', 'mushrooms']
```

Five minutes later, you realize that you really want onions on the pizza. Based on what we know so far, Domino’s would have to build an entirely new list to add onions:

```python
>>> pizza2 = pizza1 + ['onions']
>>> pizza2
['cheese', 'mushrooms', 'onions']
>>> pizza1 # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all Domino’s had to do was add onions on top of `pizza1` instead of making an entirely new `pizza2`.

Python actually allows you to mutate some objects, including lists and dictionaries. Mutability means that the object’s contents can be changed. So instead of building a new `pizza2`, we can use `pizza1.append('onions')`. Now `pizza1` would be

```python
>>> pizza1.append('onions')
>>> pizza1
['cheese', 'mushrooms', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are immutable, meaning they cannot be changed once they are created.
1.1 What Would Python Output?

Consider the following definitions and assignments and determine what Python would output for each of the calls below if they were evaluated in order. Draw the box and pointers diagrams to the right in order to keep track of the state.

1. ```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst2 is lst1
``` 

2. ```
>>> lst1.append(4)
>>> lst1
``` 

3. ```
>>> lst2
``` 

4. ```
>>> lst2[1] = 42
>>> lst2
``` 

5. ```
>>> lst1
``` 

6. ```
>>> lst1 = lst1 + [5]
>>> lst1
``` 

7. ```
>>> lst2
``` 

8. ```
>>> lst2 is lst1
```
2 List Methods

List methods are functions that are bound to a specific list. They’re called using dot notation, in the form `lst.method()`. Some common list methods:

- `lst.append(el)` mutates `lst` to add `el` to the end
- `lst.insert(i, el)` mutates `lst` to add `el` at index `i`
- `lst.sort()` mutates `lst` to sort elements in place
- `lst.remove(el)` mutates `lst` to remove the first occurrence of `el` in `lst`. If `el` is not in `lst`, an error will be thrown.
- `lst.index(el)` returns the index of the first occurrence of `el` in `lst`. If `el` is not in `lst`, an error will be thrown. This method does not mutate `lst`.

None of the mutating list methods return a new list — they simply modify the original list and return `None`.

2.1 List Mutation Questions

1. Write a function that removes all instances of `el` from `lst`.

   ```python
def remove_all(el, lst):
      """Removes all instances of el from lst.
      >>> x = [3, 1, 2, 1, 5, 1, 1, 7]
      >>> remove_all(1, x)
      >>> x
      [3, 2, 5, 7]
      ""
   ```

2. Write a function `square_elements` which takes a `lst` and replaces each element with the square of that element. *Mutate `lst` rather than returning a new list.*

   ```python
def square_elements(lst):
      """Squares every element in lst.
      >>> lst = [1, 2, 3]
      >>> square_elements(lst)
      >>> lst
      [1, 4, 9]
      ""
   ```
2.2 Extra Practice

1. Write a function which takes in a list `lst`, and two values `x` and `y`, and adds as many `ys` to the end of `lst` as there are `xs`. Do not use the `count` list method.

   ```python
   def add_this_many(x, y, lst):
       """Adds y to the end of lst the number of times x occurs.
       >>> lst = [1, 2, 4, 2, 1]
       >>> add_this_many(1, 5, lst)
       >>> lst
       [1, 2, 4, 2, 1, 5, 5]
       """
   ```

2. Write a function which reverses a list using mutation. Don’t use the `reverse` list method.

   ```python
   def reverse_list(lst):
       """Reverses lst in-place (mutating the original list).
       >>> lst = [1, 2, 3, 4]
       >>> reverse_list(lst)
       >>> lst
       [4, 3, 2, 1]
       >>> pi = [3, 1, 4, 1, 5]
       >>> reverse_list(pi)
       >>> pi
       [5, 1, 4, 1, 3]
       """
   ```
As you saw in lab, dictionaries are data structures which map keys to values. Dictionaries in Python are unordered, unlike real-world dictionaries — in other words, key-value pairs are not arranged in the dictionary in any particular order. Let’s look at an example:

```python
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'ditto': 25, 'mew': 151}
```

The keys of a dictionary can be any immutable value, such as numbers, strings, and tuples. Dictionaries themselves are mutable; we can add, remove, and change entries after creation. There is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed.

To access the value of dictionary at key, use the syntax

```python
dictionary[key]
```

Element selection and reassignment work similarly to sequences, except the square brackets contain the key, not an index.

## 3.1 What Would Python Output?

Assume these commands are entered in order after the above code has been executed in the interpreter.

1. ```python
>>> 'mewtwo' in pokemon
```

2. ```python
>>> len(pokemon)
```
3. >>> pokemon['ditto'] = pokemon['jolteon']
   >>> pokemon[('diglett', 'diglett', 'diglett')] = 51
   >>> pokemon[25] = 'pikachu'
   >>> pokemon

4. >>> pokemon['mewtwo'] = pokemon['mew'] * 2
   >>> pokemon

5. pokemon[['firetype', 'flying']] = 146

Note that the last example demonstrates that dictionaries cannot use other mutable data structures as keys. However, dictionaries can be arbitrarily deep, meaning the values of a dictionary can be themselves dictionaries.

- To add val corresponding to key or to replace the current value of key with val:
  
  `dictionary[key] = val`

- To iterate over a dictionary’s keys:
  
  `for key in dictionary: # OR for key in dictionary.keys()
     do_stuff()`

- To iterate over a dictionary’s values:
  
  `for value in dictionary.values():
     do_stuff()`

- To iterate over a dictionary’s keys and values:
  
  `for key, value in dictionary.items():
     do_stuff()`

- To remove an entry in a dictionary:
  
  `del dictionary[key]`

- To get the value corresponding to key and remove the entry:
  
  `dictionary.pop(key)`
3.2 Dictionary Questions

1. Given a dictionary \( d \), replace all occurrences of \( x \) as a value (not a key) with \( y \).

   ```python
def replace_all(d, x, y):
    """
    >>> d = {'foo': 2, 'bar': 3, 'garply': 3, 'xyzzy': 99}
    >>> replace_all(d, 3, 'poof')
    >>> d
    {'foo': 2, 'bar': 'poof', 'garply': 'poof', 'xyzzy': 99}
    """
   ```

3.3 Extra Practice

1. Given an arbitrarily deep dictionary \( d \), replace all occurrences of \( x \) as a value (not a key) with \( y \). Hint: You will need to combine iteration and recursion.

   ```python
def replace_all(d, x, y):
    """
    >>> d = {1: {2: 3, 3: 4}, 2: {4: 4, 5: 3}}
    >>> replace_all(d, 3, 1)
    >>> d
    {1: {2: 1, 3: 4}, 2: {4: 4, 5: 1}}
    """
   ```
2. Given a (non-nested) dictionary `d`, write a function which deletes all occurrences of `x` as a value. You cannot delete items in a dictionary as you are iterating through it.

```python
def remove_all(d, x):
    
    >>> d = {1:2, 2:3, 3:2, 4:3}
    >>> remove_all(d,2)
    >>> d
    {2: 3, 4: 3}
    
    """
```
The `nonlocal` keyword can be used to modify a variable in the parent frame outside the current frame (as long as it’s not the global frame). For example, consider `make_step`, which uses `nonlocal` to modify `num`:

```python
def make_step(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
```

### 4.1 Nonlocal Environment Diagrams

1. Draw the environment diagram for the following series of calls after `make_step` has been defined:

   ```
   >>> s = make_step(3)
   >>> s()
   >>> s()
   ```
2. Given the definition of `make_wallet` below, draw the environment diagram.

```python
def make_wallet(total_gold):
    def buy(cost):
        nonlocal total_gold
        if total_gold < cost:
            return 'Go farm some more champions'
        total_gold = total_gold - cost
        return total_gold
    return buy

entropy, zeal, gold = 2700, 1100, 3800
wallet = make_wallet(gold - 1000)
wallet(zeal)
wallet(entropy)
```
4.2 Nonlocal Misconceptions

For each of the following pieces of code, explain what’s wrong with the use of nonlocal.

1. a = 5
   ```python
def add_one(x):
    nonlocal x
    x += 1

>>> add_one(a)
```

2. ```python
def another_add_one():
    nonlocal a
    a += 1

>>> another_add_one(a)
```