Worksheet 5:

List Splicing, Nonlocal, Dictionaries

List Splicing

We can take subsets of lists using something called splicing.

Let's take the list:

```
>>> lst = [1, 2, 3, 4]
```

If we wanted to get only the second value to the end, we could SPLICE it by doing

```
>>> lst[1:]
```

What are we doing here?

We are telling Python to make a NEW LIST, taking the items of **lst** from INDEX 1 to the end.

This would output

[2, 3, 4]

```
>>> lst[1:3]
```

List splicing, similar to range, does NOT use the end index. This example would take the items from index 1 TILL index 3, not including index 3.

It would output

[2, 3]

We can do even more cool things with list splicing. For example, we can get every N value of a list. Let's look at a longer list example:

```
>>> lst2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> lst2[1:8:2]
```

In this case, we have a THIRD argument for our list splicing. This represents every Nth digit in the range specified. So, in this example, we want every 2nd digit in the range from numbers 2 through 8. Thus, it would output:

[2, 4, 6, 8]

Lastly, we can have NEGATIVE list splicing! This may seem complicated, but if we follow the same idea of list splicing we used for positive indexes, it becomes simpler.

Let's make a chart using **lst2**

Value	1	2	3	4	5	6	7	8	9	10
Positive index	0	1	2	3	4	5	6	7	8	9
Negative index	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Think of the negative index as ANOTHER NAME for the positive index.

```
>>> lst2[3:7]
```

As we know, this would output

[4, 5, 6, 7]

However, we could ALSO write this same list splice as

```
>>> lst2[-7:-3]
```

It uses the corresponding negative index, and uses the same rules we defined earlier for list splicing.

Now what do you think the following would output?

```
>>> lst2[-7:7]
```

It would ALSO output

[4, 5, 6, 7]

Python sees the -7 index as its starting index, and 7 as its ending index. It then looks for their corresponding values in the list, and splices based on that information!

Let's do some practice!

Practice

```
>>> my = ["oh", "my", 1, 5, 0, 89, 123]
>>> my[-2:-1]


>>> my[1:4]


>>> my[my[3:-1][1]] = ["mamma", "mia"]
>>> my[my[-5:-4][0]-1][-1:]


>>> my[0][::2]
```

Dictionaries

We can correspond different values to other values using dictionaries. Dictionaries are similar to lists, but are vastly different.

We use dictionaries to map KEYS to VALUES.

(Tip: Dictionaries are unordered)

For example, we might define a dictionary as the following:

```
>>> singers = {"Taylor Swift" : True, "Justin Bieber" : False, "Michael Jackson" : True}
```

This dictionary maps certain singers (in the form of a string) to a boolean value (whether or not I like them).

It is important to remember that we cannot have more than one key of the same value.

It is also important to remember that the KEYS of dictionaries must be IMMUTABLE values, including numbers, strings, or tuples. Dictionaries themselves are mutable however.

How can we manipulate the contents of the dictionary?

```
>>> singers["Vance Joy"] = True
>>> singers
{'Taylor Swift': True, 'Justin Bieber': False, 'Michael Jackson': True,
'Vance Joy': True}
>>> singers["Taylor Swift"] = False
>>> singers
{'Taylor Swift': False, 'Justin Bieber': False, 'Michael Jackson': True,
'Vance Joy': True}
```

Notice the second example where we replaced the value for "Taylor Swift". If we try to add a value to a key that already exists in the dictionary, the value will be replaced, since we don't allow duplicates.

Here are some useful operations on dictionaries

```
>>> list(singers.values())
[False, False, True, True]

>>> list(singers.keys())
['Taylor Swift', 'Justin Bieber', 'Michael Jackson', 'Vance Joy']

>>> list(singers.items())
[('Taylor Swift', False), ('Justin Bieber', False), ('Michael Jackson',
True), ('Vance Joy', True)]

>>> singers.pop("Michael Jackson")
True

>>> singers
{'Taylor Swift': False, 'Justin Bieber': False, 'Vance Joy': True}

>>> del singers["Justin Bieber"]
>>> singers
{'Taylor Swift': False, 'Vance Joy': True}
```

Notice that `list(singers.items())` gives the key-value pairs in the form of a TUPLE.

Practice

WWPD

```
>>> d = {"Starbucks" : 5, "Yali's" : 3, "Qualcomm" : 3, "1951" : 4}
>>> for i in d.values():
        i -= 1
>>> d





>>> for k in d:
        print(k)





>>> for k in d:
        d[k] -= 1
>>> d





>>> d[2] = "Cafe Strada"
>>> for k in d:
        d[k] += 1
>>> d




>>> d["Yali's"] = "5"
>>> for i in d.values():
        print(i)
```

Complete the function

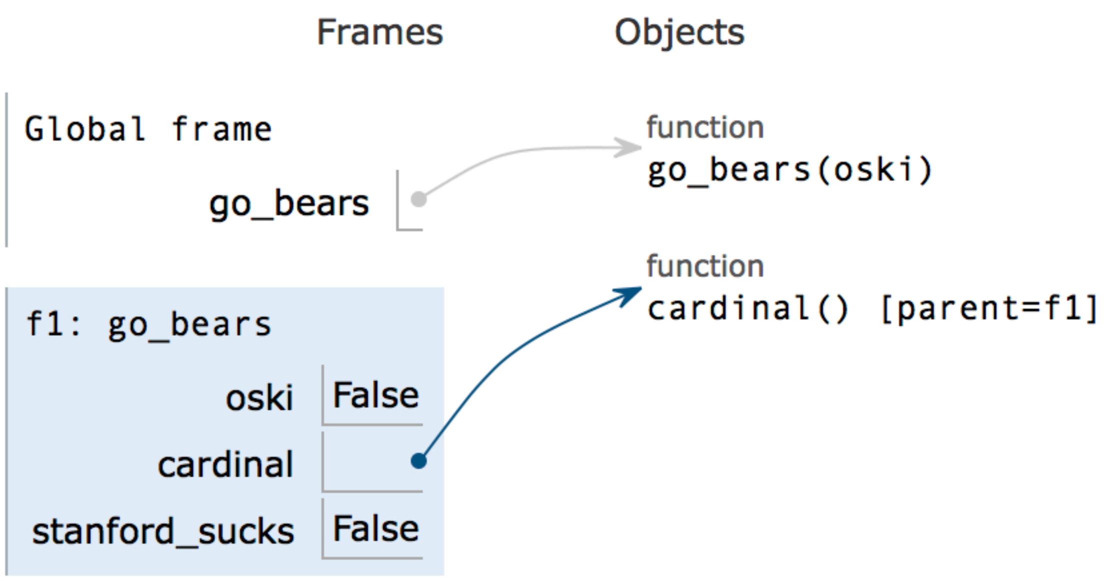
```
def switcheroo(d):
        """Switch all the keys and values in a dictionary. Assume all values
are immutable. Return a new dictionary.
        >>> d = {"a" : 1, "b" : 2, "c" : 3}
        >>> new_d = switcheroo(d)
        >>> new_d
        {1 : "a", 2 : "b", 3 : "c"}
        >>> empty = {}
        >>> new_empty = switcheroo(empty)
        >>> new_empty
        {}
        """
    _________________ = _______________________

    keys, values = ________________________, _________________________

    for ___________ in ____________________________:

                __________________________________________

    return ___________________________________
```

Nonlocal

Nonlocal statements are powerful tools that can be used in higher order functions. If we wanted to change something in our parent frame, we can use nonlocal to enable us to do so. Let's look at an example using environment diagrams.

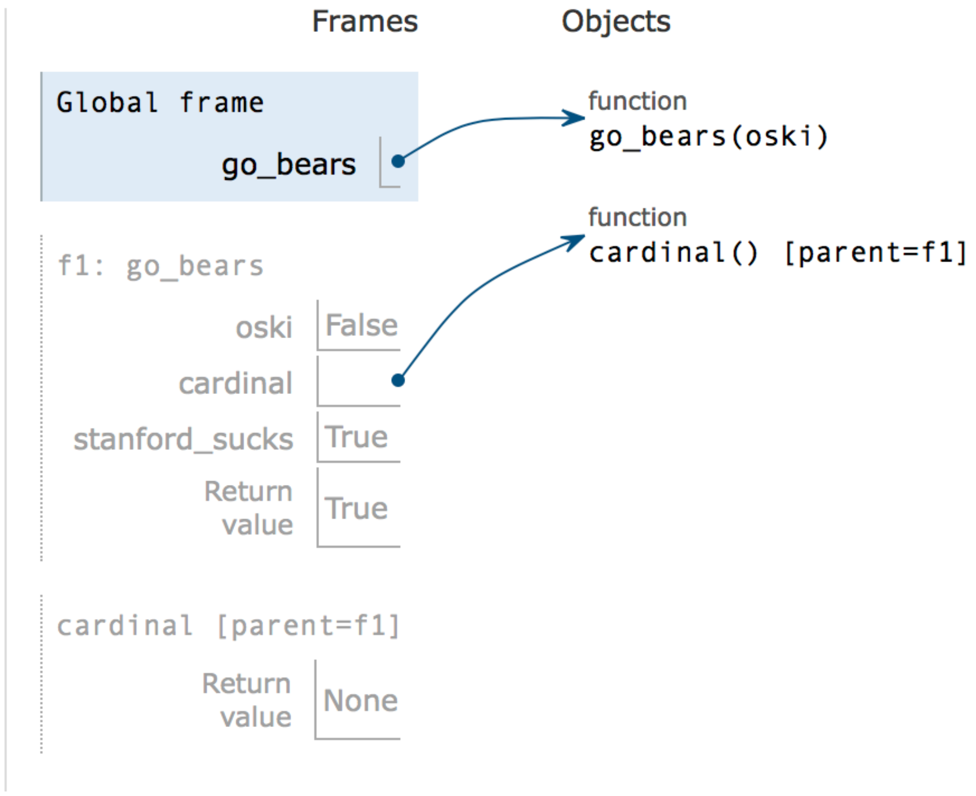
```
def go_bears(oski):
        stanford_sucks = oski
        def cardinal():
                nonlocal stanford_sucks
                stanford_sucks = not stanford_sucks
        cardinal()
        return stanford_sucks
go_bears(False)
```

If we build an environment diagram, up until the return statement in go_bears, we get something like this.



When we call the cardinal function, we assign the variable *stanford_sucks* to be nonlocal. This allows us to edit the variable *stanford_sucks*, which exists in f1, inside f2.

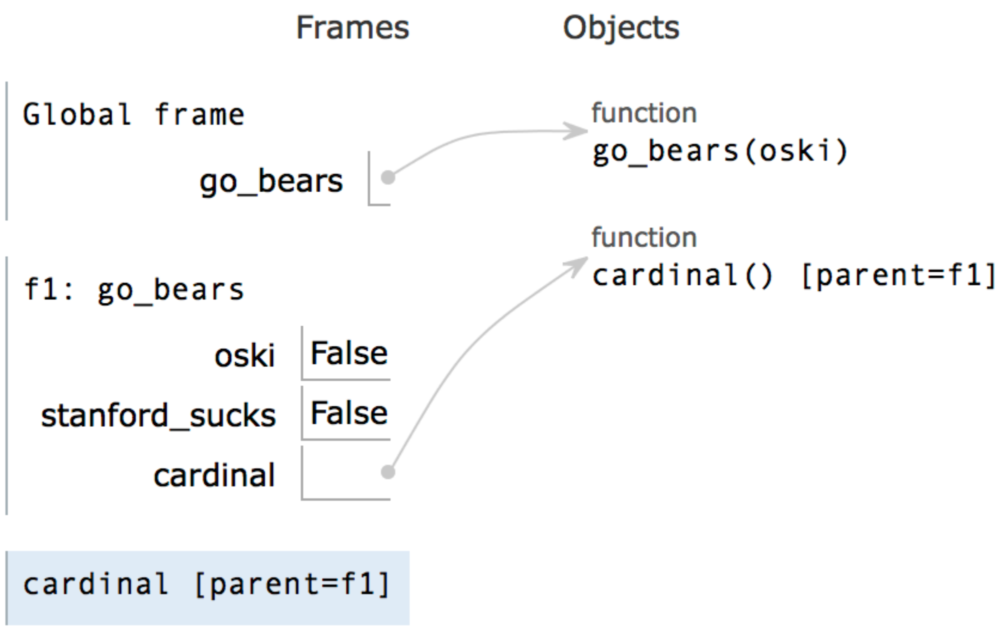
At the end of execution, the environment diagram would look like:



As you can see, *stanford_sucks* changed inside f1.

To better see this concept, let's see what happens if we didn't have the nonlocal statement.

```
def go_bears(oski):
        stanford_sucks = oski
        def cardinal():
                #nonlocal stanford_sucks
                stanford_sucks = not stanford_sucks
        cardinal()
        return stanford_sucks
go_bears(False)
```



When we try to assign *stanford_sucks* inside f2, we get an error, as Python is confused whether to change the variable in f1, or make a new variable.

Some important edge case:

We cannot change global variables using nonlocal and we cannot change variables inside the

Let's do some practice to investigate the power of nonlocal statements.

Practice

Draw an environment diagram for the following:

```
def dan (garcia):
        cs10 = 61
        61a = garcia * 4
        def professor(61c):
                nonlocal professor
                professor = lambda a : 61
                return 61c + professor(cs10)
        return professor(61a)
garcia = dan(10)
```

Complete the function

```
def memory(n):
"""
        A function that takes in a value x and updates and prints the result
        based on input functions. (Credits to 61A discussion worksheet)
    >>> f = memory(10)
    >>> f = f(lambda x: x * 2)
    20
    >>> f = f(lambda x: x - 7)
    13
    >>> f = f(lambda x: x > 5)
    True
        """
```