NONLOCAL, ITERATORS AND GENERATORS

COMPUTER SCIENCE MENTORS

March 1, 2021 to March 3, 2021

1 Nonlocal

For this semester, there won't be extensive nonlocal coding questions, but still go over this short blurb and try to understand our reverse-environment diagram question.

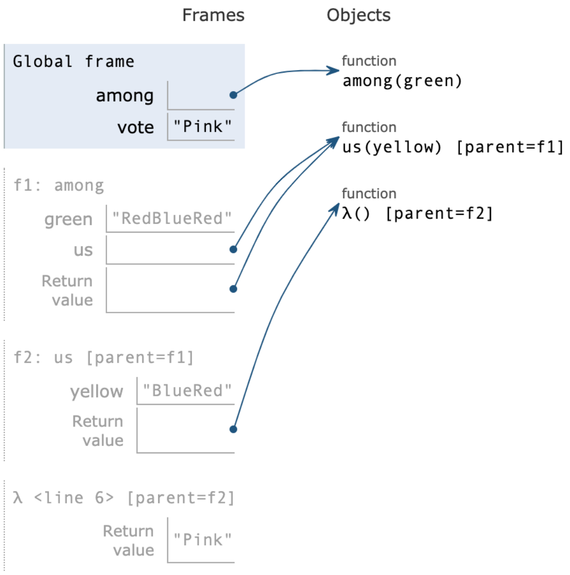
The first time we assign a value to a `nonlocal` variable, rather than declare a new variable in the current frame, we bind the value to the variable in the first parent frame that contains such a variable. The variable does not exist in the current frame!
Note: you cannot declare variables in the global frame as `nonlocal`.

```
def example_without_nonlocal():
    grade = 1.0
    def gpa_boost():
        grade = 4.0 # creates a variable named grade
    gpa_boost()
    print(grade)
>>> example_without_nonlocal()
1.0

def example_with_nonlocal():
    grade = 1.0
    def gpa_boost():
        nonlocal grade
        grade = 4.0 # modifies the variable in the
                    # example_with_nonlocal frame
    gpa_boost()
    print(grade)
>>> example_with_nonlocal()
4.0
```

1. **among us**
   Fill in each blank in the code example below so that its environment diagram is the following. You do not need to use all the blanks.

```
def among(green):
    def us(yellow):
        ______________________
        yellow += ___________
        green += ____________
        ______________________
        return ______________
    return ______________
vote = among('Red')('Blue')()
```



CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

2 Iterators and Generators

An **iterable** is any container that can be processed sequentially. Think of an iterable as anything you can loop over, such as lists or strings. You can see this in **for** loops, which sequentially loop through each element of a sequence. The anatomy of the for loop can be described as:

```
for some_var in iterable:
    <do something with some_var>
```

An **iterator** remembers where it is during its iteration. Though an iterator is an iterable, the reverse is not necessarily true. Think of an iterable as a book whereas an iterator is a bookmark.

Generators, which are a specific type of **iterators**, are created using the traditional function definition syntax in Python (**def**) with the body of the function containing one or more `yield` statements. When a generator (a function that has `yield` in the body) is called, it returns a generator object. When we call the generator object, we evaluate the body of the function until we have yielded a value. The `yield` statement pauses the function, yields the value, saves the local state so that evaluation can be resumed right where it left off. `yield` operates similarly to a return statement.

1. Given the following code block, what is outputted by the lines that follow?

```
def foo():
    a = 0
    if a == 0:
        print("Hello")
        yield a
        print("World")
```

```
>>> foo()
```

```
>>> foo_gen = foo()
>>> next(foo_gen)
```

```
>>> next(foo_gen)
```

```
>>> for i in foo():
...   print(i)
```

2. How can we modify `foo` so that it satisfies the following doctests?

```
>>> a = list(foo())
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

3. Define `filter_gen`, a generator that takes in iterable `s` and one-argument function `f` and yields every value from `s` for which `f` returns a truthy value.

```
def filter_gen(s, f):
    """
    >>> list(filter_gen([1, 2, 3, 4, 5],
                                    lambda x: x % 2 == 0))
    [2, 4]
    >>> list(filter_gen((1, 2, 3, 4, 5), lambda x: x < 3))
    [1, 2]
    """
```

4. Define `all_sums`, a generator that iterates through all the sums that can be formed by adding the elements in `lst`.

```
def all_sums(lst):
    """
    >>> gen = all_sums([1, 2, 3])
    >>> sorted(gen)
    [0, 1, 2, 3, 3, 4, 5, 6]
    """
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni

3 Extra Practice: Trees + Generators

1. Define `tree_sequence`, a generator that iterates through a tree by first yielding the root value and then yielding the values from each branch.

```
def tree_sequence(t):
    """
    >>> t = tree(1, [tree(2, [tree(5)]), tree(3, [tree(4)])])
    >>> print(list(tree_sequence(t)))
    [1, 2, 5, 3, 4]
    """
```

CSM 61A Spring 2021: Ivan Penev and Joshua Baum, with
Jamie Ip, Kenneth Chi, Kevin Moy, Mathilde McKeever, Cyrus Bugwadia, Peter Zhang, and
Kunal Agarwal, Luke Liu, John So, Jemmy Zhou, Ruslana Yurtyn, Vivian Lu, Matt Au, Lily Yang, Mihira Patel, Jessica Yu, Aidan Tong, Matthew Guo, Nikhita Anumukonda, Laryn Qi, Grace Yi, Jennifer Huang, Cindy Lin, James Fletcher, Sean Sananikone, Abinaya Srikant, Uma Unni