

NONLOCAL AND ITERATORS

COMPUTER SCIENCE MENTORS

October 12 to October 15, 2020

1 Nonlocal

To evaluate a variable name, we find its value from the first frame (current frame, parent, parent's parent, ...) the variable is defined. The first time we assign (bind) a value to a variable, we declare a new variable in the current frame and bind to the respective value.

The `nonlocal` keyword introduces more control over this process. 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 of the same name found in the first parent frame that contains such a variable. This means that this 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 in the
                    # gpa_boost frame
    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 instead
                    # of creating a new variable

    gpa_boost()
    print(grade)
>>> example_with_nonlocal()
4.0
```

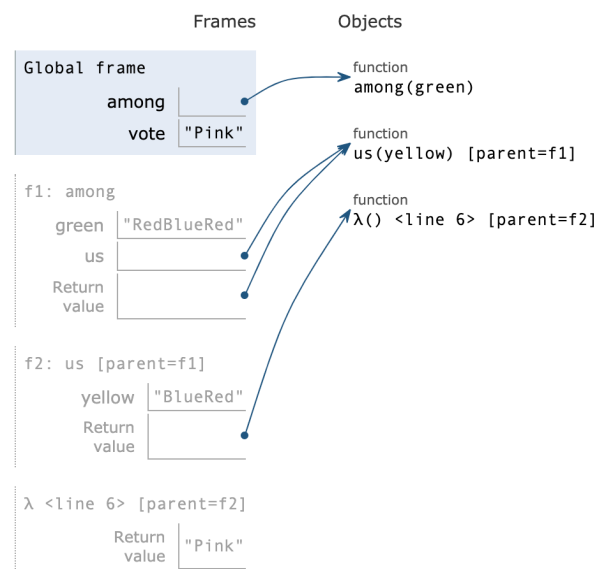
1. among us

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')()
```



2. Pingpong again...

Time for some more ping-pong! Remember, the ping-pong sequence counts up starting from 1 and is always either counting up or counting down. At element k , the direction switches if k is a multiple of 7 or contains the digit 7.

The first 20 elements of the ping-pong sequence are listed below, with direction swaps marked using brackets at the 7th, 14th, and 17th elements

1 2 3 4 5 6 [7] 6 5 4 3 2 1 [0] 1 2 [3] 2 1 0

Implement a function `make_pingpong_tracker` that returns the next value in the pingpong sequence each time it is called. You may use assignment statements.

```
def has_seven(k): # Use this function for your answer below
    if k % 10 == 7:
        return True
    elif k < 10:
        return False
    else:
        return has_seven(k // 10)

def make_pingpong_tracker():
    """ Returns a function that returns the next value in the
    pingpong sequence each time it is called.
    >>> output = []
    >>> x = make_pingpong_tracker()
    >>> for _ in range(9):
    ... output += [x()]
    >>> output
    [1, 2, 3, 4, 5, 6, 7, 6, 5]
    """
    index, current, add = 1, 0, True
    def pingpong_tracker():
        if add:
            current += 1
        else:
            current -= 1
        if has_seven(current):
            add = not add
        index += 1
    return current
```

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 `True`

```
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 `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]  
    """
```