# Instructions

---

> Please download lab materials lab05.zip from our QQ group if you don't have one.

---

In this lab, you have two tasks:

- Think about what would Python display in section 3 and verify your answer with Ok.
- Complete the required problems described in section 4 and submit your code to our OJ website.

The starter code for these problems is provided in lab05.py.

**Submission**: As instructed before, you need to submit your work with Ok by `python ok --submit`. You may submit more than once before the deadline, and your score of this assignment will be the highest one of all your submissions.

**Readings**: You might find the following reference to the textbook useful:

- Section 2.4
- Section 4.2

# Review

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the next section and refer back here when you get stuck.

# Nonlocal

We say that a variable defined in a frame is *local* to that frame. A variable is **nonlocal** to a frame if it is defined in the environment that the frame belongs to but not the frame itself, i.e. in its parent or ancestor frame.

So far, we know that we can access variables in parent frames:

```
def make_adder(x):
    """ Returns a one-argument function that returns the result of
    adding x and its argument. """
    def adder(y):
        return x + y
    return adder
```

Here, when we call `make_adder`, we create a function `adder` that is able to look up the name `x` in `make_adder`'s frame and use its value.

However, we haven't been able to `modify` variable in parent frames. Consider the following function:

```
def make_withdraw(balance):
    """Returns a function which can withdraw
    some amount from balance

    >>> withdraw = make_withdraw(50)
    >>> withdraw(25)
    25
    >>> withdraw(25)
    0
    """
    def withdraw(amount):
        if amount > balance:
            return "Insufficient funds"
        balance = balance - amount
        return balance
    return withdraw
```

The inner function `withdraw` attempts to update the variable balance in its parent frame. Running this function's doctests, we find that it causes the following error:

```
UnboundLocalError: local variable 'balance' referenced before assignment
```

Why does this happen? When we execute an assignment statement, remember that we are either creating a new binding in our current frame or we are updating an old one in the current frame. For example, the line `balance = ...` in `withdraw`, is creating the local variable balance inside `withdraw`'s frame. This assignment statement tells Python to expect a variable called balance inside `withdraw`'s frame, so Python will not look in parent frames for this variable. However, notice that we tried to compute `balance - amount` before the local variable was created! That's why we get the `UnboundLocalError`.

To avoid this problem, we introduce the `nonlocal` keyword. It allows us to update a variable in a parent frame!

---

> Some important things to keep in mind when using `nonlocal`
>
> - `nonlocal` cannot be used with global variables (names defined in the global frame). If no `nonlocal` variable is found with the given name, a `SyntaxError` is raised. A name that is already local to a frame cannot be declared as `nonlocal`.

---

Consider this improved example:

```
def make_withdraw(balance):
    """Returns a function which can withdraw
    some amount from balance

    >>> withdraw = make_withdraw(50)
    >>> withdraw(25)
    25
    >>> withdraw(25)
    0
    """
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return "Insufficient funds"
        balance = balance - amount
        return balance
    return withdraw
```

The line `nonlocal balance` tells Python that balance will not be local to this frame, so it will look for it in parent frames. Now we can update `balance` without running into problems.

# Iterators

An iterable is any object that can be iterated through, or gone through one element at a time. One construct that we've used to iterate through an iterable is a for loop:

```
for elem in iterable:
    # do something
```

`for` loops work on any object that is *iterable*. We previously described it as working with any sequence -- all sequences are iterable, but there are other objects that are also iterable! We define an **iterable** as an object on which calling the built-in function `iter` function returns an *iterator*. An iterator is another type of object that allows us to iterate through an iterable by keeping track of which element is next in the sequence.

To illustrate this, consider the following block of code, which does the exact same thing as the for statement above:

```
iterator = iter(iterable)
try:
    while True:
        elem = next(iterator)
        # do something
except StopIteration:
    pass
```

Here's a breakdown of what's happening:

- First, the built-in `iter` function is called on the iterable to create a corresponding *iterator*.
- To get the next element in the sequence, the built-in `next` function is called on this iterator.
- When `next` is called but there are no elements left in the iterator, a `StopIteration` error is raised. In the for loop construct, this exception is caught and execution can continue.

Calling `iter` on an iterable multiple times returns a new iterator each time with distinct states (otherwise, you'd never be able to iterate through a iterable more than once). You can also call `iter` on the iterator itself, which will just return the same iterator without changing its state. However, note that you cannot call next directly on an iterable.

Let's see the `iter` and `next` functions in action with an iterable we're already familiar with -- a list.

```
>>> lst = [1, 2, 3, 4]
>>> next(lst)             # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> list_iter
<list_iterator object ...>
>>> next(list_iter)       # Calling next on an iterator
1
>>> next(list_iter)       # Calling next on the same iterator
2
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
3
>>> list_iter2 = iter(lst)
>>> next(list_iter2)      # Second iterator has new state
1
>>> next(list_iter)       # First iterator is unaffected by second iterator
4
>>> next(list_iter)       # No elements left!
StopIteration
>>> lst                   # Original iterable is unaffected
[1, 2, 3, 4]
```

Since you can call `iter` on iterators, this tells us that they are also iterables! Note that while all iterators are iterables, the converse is not true - that is, not all iterables are iterators. You can use iterators wherever you can use iterables, but note that since iterators keep their state, they're only good to iterate through an iterable once:

```
>>> list_iter = iter([4, 3, 2, 1])
>>> for e in list_iter:
...     print(e)
4
3
2
1
>>> for e in list_iter:
...     print(e)
```

---

> **Analogy**: An iterable is like a book (one can flip through the pages) and an iterator for a book would be a bookmark (saves the position and can locate the next page). Calling `iter` on a book gives you a new bookmark independent of other bookmarks, but calling `iter` on a bookmark gives you the bookmark itself, without changing its position at all. Calling `next` on the bookmark moves it to the next page, but does not change the pages in the book. Calling `next` on the book wouldn't make sense semantically. We can also have multiple bookmarks, all independent of each other.

---

# Iterable Uses

We know that lists are one type of built-in iterable objects. You may have also encountered the `range(start, end)` function, which creates an iterable of ascending integers from start (inclusive) to end (exclusive).

```
>>> for x in range(1, 6):
...     print(x)
...
1
2
3
4
```

Ranges are useful for many things, including performing some operations for a particular number of iterations or iterating through the indices of a list.

There are also some built-in functions that take in iterables and return useful results:

- `map(f, iterable)` - Creates iterator over `f(x)` for each `x` in `iterable`
- `filter(f, iterable)` - Creates iterator over `x` for each `x` in `iterable` if `f(x)`
- `zip(iter0, iter2)` - Creates iterator over co-indexed pairs `(x, y)` from both input iterables
- `reversed(iterable)` - Creates iterator over all the elements in the input iterable in reverse order
- `list(iterable)` - Creates a list containing all the elements in the input iterable
- `tuple(iterable)` - Creates a tuple containing all the elements in the input iterable
- `sorted(iterable)` - Creates a sorted list containing all the elements in the input iterable

# Generators

We can create our own custom iterators by writing a *generator* function, which returns a special type of iterator called a **generator**. Generator functions have `yield` statements within the body of the function instead of `return` statements. Calling a generator function will return a generator object and will *not* execute the body of the function.

For example, let's consider the following generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

Calling `countdown(k)` will return a generator object that counts down from `k` to `0`. Since generators are iterators, we can call `iter` on the resulting object, which will simply return the same object. Note that the body is not executed at this point; nothing is printed and no numbers are output.

```
>>> c = countdown(5)
>>> c
<generator object countdown ...>
>>> c is iter(c)
True
```

So how is the counting done? Again, since generators are iterators, we call `next` on them to get the next element! The first time `next` is called, execution begins at the first line of the function body and continues until the `yield` statement is reached. The result of evaluating the expression in the `yield` statement is returned. The following interactive session continues from the one above.

```
>>> next(c)
Beginning countdown!
5
```

Unlike functions we've seen before in this course, generator functions can remember their state. On any consecutive calls to `next`, execution picks up from the line after the `yield` statement that was previously executed. Like the first call to `next`, execution will continue until the next `yield` statement is reached. Note that because of this, `Beginning countdown!` doesn't get printed again.

```
>>> next(c)
4
>>> next(c)
3
```

The next 3 calls to `next` will continue to `yield` consecutive descending integers until `0`. On the following call, a `StopIteration` error will be raised because there are no more values to `yield` (i.e. the end of the function body was reached before hitting a `yield` statement).

```
>>> next(c)
2
>>> next(c)
1
>>> next(c)
0
>>> next(c)
Blastoff!
StopIteration
```

Separate calls to `countdown` will create distinct generator objects with their own state. Usually, generators shouldn't restart. If you'd like to reset the sequence, create another generator object by calling the generator function again.

```
>>> c1, c2 = countdown(5), countdown(5)
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
5
>>> next(c2)
Beginning countdown!
5
```

Here is a summary of the above:

- A generator function has a `yield` statement and returns a *generator object*.
- Calling the `iter` function on a generator object returns the same object without modifying its current state.
- The body of a generator function is not evaluated until `next` is called on a resulting generator object. Calling the `next` function on a generator object computes and returns the next object in its sequence. If the sequence is exhausted, `StopIteration` is raised.
- A generator "remembers" its state for the next `next` call. Therefore,
    - The first `next` call works like this:
        - Enter the function and run until the line with `yield`.

        - Return the value in the `yield` statement, but remember the state of the function for future `next` calls.
    - And subsequent `next` calls work like this:
        - Re-enter the function, start at **the line after the `yield` statement that was previously executed**, and run until the next `yield` statement.
        - Return the value in the `yield` statement, but remember the state of the function for future `next` calls.
- Calling a generator function returns a brand new generator object (like calling `iter` on an iterable object).
- A generator should not restart unless it's defined that way. To start over from the first element in a generator, just call the generator function again to create a new generator.

Another useful tool for generators is the `yield from` statement (introduced in Python 3.3). `yield from` will yield all values from an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2, 3, 4])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
4
>>> next(g)
StopIteration
```

# What Would Python Display?

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python ok -q wwpd -u
```

**Important**: Enter `StopIteration` if a `StopIteration` exception occurs, `Error` if you believe a different error occurs, and `Iterator` if the output is an iterator object.

```
>>> s = [1, 2, 3, 4]
>>> t = iter(s)
>>> next(s)
------

>>> next(t)
------

>>> next(t)
------

>>> iter(s)
------

>>> next(iter(s))
------

>>> next(iter(t))
------

>>> next(iter(s))
------

>>> next(iter(t))
------

>>> next(t)
------
```

```
>>> r = range(6)
>>> r_iter = iter(r)
>>> next(r_iter)
------

>>> [x + 1 for x in r]
------

>>> [x + 1 for x in r_iter]
------

>>> next(r_iter)
------

>>> list(range(-2, 4))   # Converts an iterable into a list
------
```

```
>>> map_iter = map(lambda x : x + 10, range(5))
>>> next(map_iter)
------

>>> next(map_iter)
------

>>> list(map_iter)
------

>>> for e in filter(lambda x : x % 2 == 0, range(1000, 1008)):
...     print(e)
------

>>> [x + y for x, y in zip([1, 2, 3], [4, 5, 6])]
------

>>> for e in zip([10, 9, 8], range(3)):
...   print(tuple(map(lambda x: x + 2, e)))
------
```

# Required Problems

In this section, you are required to complete the problems below and submit your code to OJ website as instructed in lab00 to get your answer scored.

# Problem 1: Count (100pts)

Implement `count`, which takes in an iterator `t` and returns the number of times the value `x` appears in the first `n` elements of `t`. A value appears in a sequence of elements if it is equal to an entry in the sequence.

---

**Note:** You can assume that `t` will have at least `n` elements.

---

```
def count(t, n, x):
    """Return the number of times that x appears in the first n elements of
iterator t.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> count(s, 10, 9)
    3
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> count(s2, 3, 10)
    2
    >>> s = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> count(s, 1, 3)
    1
    >>> count(s, 4, 2)
    3
    >>> next(s)
    2
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> count(s2, 6, 6)
    2
    """
    "*** YOUR CODE HERE ***"
```

# Problem 2: Repeated (100pts)

Implement `repeated`, which takes in an iterator `t` and returns the first value in `t` that appears `k` times in a row.

---

> **Note:** You can assume that the iterator `t` will have a value that appears at least `k` times in a row. If you are receiving a `StopIteration`, your `repeated` function is likely not identifying the correct value.

---

Your implementation should iterate through the items in a way such that if the same iterator is passed into `repeated` twice, it should continue in the second call at the point it left off in the first. An example of this behavior is in the doctests.

```python
def repeated(t, k):
    """Return the first value in iterator T that appears K times in a row.
    Iterate through the items such that if the same iterator is passed into
    the function twice, it continues in the second call at the point it left
    off in the first.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s, 1)
    10
    >>> repeated(s, 2)
    9
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s2, 3)
    8
    >>> s = iter([3, 2, 2, 2, 2, 2, 2, 4, 4, 5, 5, 5])
    >>> repeated(s, 3)
    2
    >>> repeated(s, 3)
    2
    >>> repeated(s, 3)
    5
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> repeated(s2, 3)
    2
    """
    assert k >= 1
    "*** YOUR CODE HERE ***"
```

# Problem 3: Scale (100pts)

Implement the generator function `scale(it, multiplier)`, which yields elements of the given iterable `it`, scaled by `multiplier`.

As an extra challenge, try writing this function using a `yield from` statement!

```
def scale(it, multiplier):
    """Yield elements of the iterable it scaled by a number multiplier.

    >>> m = scale(iter([1, 5, 2]), 5)
    >>> type(m)
    <class 'generator'>
    >>> list(m)
    [5, 25, 10]
    >>> # generators allow us to represent infinite sequences!!!
    >>> def naturals():
    ...     i = 0
    ...     while True:
    ...         yield i
    ...         i += 1
    >>> m = scale(naturals(), 2)
    >>> [next(m) for _ in range(5)]
    [0, 2, 4, 6, 8]
    """
    "*** YOUR CODE HERE ***"
```

# Problem 4: Merge (100pts)

Write a generator function `merge` that takes in two infinite generators `a` and `b` that are in increasing order without duplicates and returns a generator that has all the elements of both generators, in increasing order, without duplicates.

---

> **Note:** You may assume that both `a` and `b` contains unlimited elements. As a result, calling `next(a)` and `next(b)` will never throw `StopIteration` exception.

---

```python
def merge(a, b):
    """Merge two generators that are in increasing order and without
duplicates.
    Return a generator that has all elements of both generators in increasing
    order and without duplicates.

    >>> def sequence(start, step):
    ...     while True:
    ...         yield start
    ...         start += step
    >>> a = sequence(2, 3) # 2, 5, 8, 11, 14, ...
    >>> b = sequence(3, 2) # 3, 5, 7, 9, 11, 13, 15, ...
    >>> result = merge(a, b) # 2, 3, 5, 7, 8, 9, 11, 13, 14, 15
    >>> [next(result) for _ in range(10)]
    [2, 3, 5, 7, 8, 9, 11, 13, 14, 15]
    """
    "*** YOUR CODE HERE ***"
```

# Problem 5: Hailstone (100pts)

Write a generator that outputs the hailstone sequence from hw01.

Here's a quick reminder of how the hailstone sequence is defined:

1. Pick a positive integer $n$ as the start.
2. If $n$ is even, divide it by 2.
3. If $n$ is odd, multiply it by 3 and add 1.
4. Continue this process until $n$ is 1.

For some extra practice, try writing a solution using recursion. Since hailstone returns a generator, you can `yield from` a call to hailstone!

```
def hailstone(n):
    """Return a generator that outputs the hailstone sequence.

    >>> for num in hailstone(10):
    ...     print(num)
    10
    5
    16
    8
    4
    2
    1
    """
    "*** YOUR CODE HERE ***"
```