1. Review of arrays as an essential built-in aggregate data type
2. Discuss "rules" when using the built-in list as an array (mostly: only use O(1) operations; exception: `append`)
3. Discuss the API for our list ADT (shocker: based on Python's sequence operations!)
4. Define goal: build a list ADT implementation based on the array as a storage container
5. Define a suitable class containing an array, and implement some basic APIs
6. Look at how "special methods" let us tie our ADT to predefined Python operators
7. Understand the concept of an iterator, and how to implement the protocol for our list ADT (using a class)
8. Look at a generator implementation of an iterator

***

# 1. Arrays

An essential built-in aggregate data type.

In Python: block of memory containing a finite number of contiguous *references* to objects.

Reading/Writing reference requires computing an offset to the reference based on an index, and is O(1)

Appending to the end of an array requires extending the array --- Python does this very efficiently; effectively ~ O(1)

# 2. Rules of list as array use

- O(1) operations are ok --- they translate trivially into memory accesses
- (empty) list creation
- read/write from/to given index
- get length
- append
- del lst[len(lst)-1] (delete from end)

Cannot use O(N) operations or any that carry out extra logic!
- e.g., insert (in middle), del (from middle), extend, slice-based indexing, sorting (via `sorted`), searching (via `in`), iteration (via `for .. in`), concatenation, pop, remove
# 3. List ADT

See Common/Mutable sequence operations; https://docs.python.org/3.5/library/stdtypes.html#common-sequence-operations, https://docs.python.org/3.5/library/stdtypes.html#mutable-sequence-types

# 4. Goal: array-based list ADT implementation

# 5. class ArrayList

```python
def __init__(self):
    self.data = []

def append(self, elem):
    pass

def elem_at(self, idx):
    pass

* * *

def extend(self, seq):
    pass
```

# 6. "Special" methods

Start with: __str__, __repr__
Subscript notation (brackets): __len__, __getitem__, __setitem__

See https://docs.python.org/3.5/reference/datamodel.html#special-method-names, https://docs.python.org/3.5/reference/datamodel.html#emulating-container-types

# 7. Iterators & Implementation

Key methods: __iter__, __next__
Built-in functions: iter(), next()
Exception: StopIteration

Basic idea: implement the __iter__ method for our ADT, which returns an "iterator object" that maintains the current state of traversing our underlying data (what happens if we modify the data structure while iterating?)

# 8. Generators, `yield`, and using them to implement Iterators
- `yield` takes the place of return ... but that's just the beginning

- `yield` keyword's presence in a function/method automatically designates it as a *generator* --- the function's body *doesn't get run when it is called*!!!

- triggered with `next` builtin function; automatically throws StopIteration when generator returns

    ```python
def mygen():
        print('Hello from mygen')
        yield 10

def mygen():
    print('First yield')
    yield 1
    print('Second yield')
    yield 2
    print('Third yield')
    yield 3
    print('Fin')

def mygen():
    i = 0
    print('First yield')
    yield i
    i += 1
    print('Second yield')
    yield i
    i += 1
    print('Third yield')
    yield i
    print('Fin')

    * * *

Raw source:

```python
class ArrayList:
    def __init__(self):
        self.data = []

    def append(self, elem):
```
def extend(self, seq):
    for x in seq:
        self.append(x)
    return self

def __len__(self):
    return len(self.data)

def __getitem__(self, key):
    if isinstance(key, slice):
        return "a slice!"
    return self.data[key]

def __setitem__(self, key, val):
    self.data[key] = val

def __str__(self):
    return str(self.data)

def __repr__(self):
    return repr(self.data)

def __delitem__(self, key):
    if key == len(self.data) - 1:
        del self.data[key]
    else:
        for i in range(key, len(self.data) - 1):
            self.data[i] = self.data[i+1]
        del self.data[len(self.data) - 1]

def __iter__(self):
    class Iterator:
        def __init__(self, data):
            self.idx = len(data)
            self.data = data
        def __iter__(self):
            return self
        def __next__(self):
            if self.idx > 0:
self.idx -= 1
return self.data[self.idx]
raise StopIteration
return Iterator(self.data)

# generator based iterator implementation
# def __iter__(self):
#     pos = 0
#     while pos < len(self.data):
#         yield self.data[pos]
#         pos += 1

def __contains__(self, item):
    for i in range(len(self.data)):
        if self.data[i] == item:
            return True
    else:
        return False