• Ready to start looking at data structure implementations!

▼ Categories of data structures:

• **Sequential** ← our first category
• Mapping / Lookup
• Ordered (vs. Unordered)
• “Linked” (vs. Array-based)

▼ Data structure: related terms

▼ Abstract data type (ADT)
• a set of functions (APIs) that allow us to access and manage a collection of data in predefined ways (e.g., as a stack or queue)

▼ Aggregate data type (err ... not “ADT”)
• one “container” for many discrete pieces of data — more of an implementation view
• note: constituent data may also be of aggregate type (i.e., this is a recursive definition)

▼ two general approaches to aggregating data:
• objects (based on classes with multiple attributes)
• arrays

▼ Arrays
• allow for positional — i.e., index-based — access to constituent elements

▼ accessing and modifying any element is a O(1) operation
• depends on random access property of memory

▼ requires that the memory address of an element can be computed from an index in O(1)
• typically implies that elements are contiguously allocated in memory, and are all the same size
• in turn implies that arrays are very efficient storage mechanisms (no overhead)

▼ disadvantages?
• member data must be homogeneous
• array occupies a monolithic chunk of memory — cannot be split up (e.g., to utilize different parts of memory), and if surrounded by other things, cannot be trivially expanded

▼ Are Python Lists = Arrays?
Simple answer: no

- Can’t be. Arrays are not ADTs (i.e., have no operations besides creation and basic indexing) — they are storage mechanisms. Lists expose plenty of abstract operations (e.g., insert, append, delete, search)

More interesting question: are they based on arrays?

One argument against: Lists can be heterogenous

- Ok! Array elements can be references to objects, and references can all be the same size.

Can use operation time complexity as clues:

- Consider: O of read/update element at some index, append, insert, delete

Timing for list operations:

- timeit calls: (ignore “MyList” for now)

```python
to_time = [ ('_ = l[random.randrange(len(l))]','l = list(range({}))'),
            ('l.append(0)','l = []'),
            ('l.insert(0, 0)','l = []'),
            ('l.prepend(0)', 'from __main__ import MyList ; l = MyList()'),
            ('del l[0]', 'l = list(range({}))') ]
```

```python
for op, setup in to_time:
    print('{}{:>10}{:>40}'.format('n', op))
    for n in range(10**4, 10**5+1, 10**4):
        t = timeit('for _ in range({})': '{}'.format(n, op),
                      setup='import random ; ' + setup.format(n),
                      number=1)
        print('{}{:>10d}{:>40.6e}'.format(n, t/n))
```

- timing output:

```
  n   _ = l[random.randrange(len(l))]
       10000    1.690114e-06
       20000    1.247375e-06
       30000    1.491479e-06
       40000    1.456922e-06
       50000    1.397985e-06
       60000    1.127639e-06
       70000    1.444051e-06
       80000    1.580476e-06
       90000    1.367383e-06
      100000    1.235515e-06

  n   l.append(0)
       10000    1.082489e-07
       20000    8.617520e-08
```
<table>
<thead>
<tr>
<th>n</th>
<th>l.insert(0, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>2.821812e-06</td>
</tr>
<tr>
<td>20000</td>
<td>5.463432e-06</td>
</tr>
<tr>
<td>30000</td>
<td>6.942879e-06</td>
</tr>
<tr>
<td>40000</td>
<td>9.100819e-06</td>
</tr>
<tr>
<td>50000</td>
<td>1.224535e-05</td>
</tr>
<tr>
<td>60000</td>
<td>1.512371e-05</td>
</tr>
<tr>
<td>70000</td>
<td>1.782433e-05</td>
</tr>
<tr>
<td>80000</td>
<td>2.071237e-05</td>
</tr>
<tr>
<td>90000</td>
<td>2.382006e-05</td>
</tr>
<tr>
<td>100000</td>
<td>2.704227e-05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>l.prepend(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>1.180755e-03</td>
</tr>
<tr>
<td>20000</td>
<td>2.462027e-03</td>
</tr>
<tr>
<td>30000</td>
<td>3.683736e-03</td>
</tr>
<tr>
<td>40000</td>
<td>4.708356e-03</td>
</tr>
<tr>
<td>50000</td>
<td>5.831405e-03</td>
</tr>
<tr>
<td>60000</td>
<td>6.960435e-03</td>
</tr>
<tr>
<td>70000</td>
<td>8.229487e-03</td>
</tr>
<tr>
<td>80000</td>
<td>9.182261e-03</td>
</tr>
<tr>
<td>90000</td>
<td>1.053582e-02</td>
</tr>
<tr>
<td>100000</td>
<td>1.259566e-02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>del l[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>1.201291e-06</td>
</tr>
<tr>
<td>20000</td>
<td>2.782344e-06</td>
</tr>
<tr>
<td>30000</td>
<td>4.441418e-06</td>
</tr>
<tr>
<td>40000</td>
<td>5.761721e-06</td>
</tr>
<tr>
<td>50000</td>
<td>7.609048e-06</td>
</tr>
<tr>
<td>60000</td>
<td>9.866580e-06</td>
</tr>
<tr>
<td>70000</td>
<td>1.141116e-05</td>
</tr>
<tr>
<td>80000</td>
<td>1.295294e-05</td>
</tr>
<tr>
<td>90000</td>
<td>1.473281e-05</td>
</tr>
<tr>
<td>100000</td>
<td>1.688042e-05</td>
</tr>
</tbody>
</table>

From “Sort timings” Mathematica Notebook (PDF on website)
Random list read timings

List append timings
Timing for list operations:

```c
self->allocated = new_allocated;
items = (PyObject **)malloc(_PyList_SIZE_Of_AllocElements(newsize));
if (new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
    /* check for integer overflow */
    {  
        PyErr_SetString(PyExc_OverflowError, "overflow in list growth pattern");  
        return -1;  
    }
}
```

* The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, system realloc().
* sequence of appends() in the presence of a poorly-performing
* enough to give linear-time amortized behavior over a long
* allocated (the C standard doesn't guarantee this, but it's hard
* fails when passed a number of bytes <= the number of bytes last
* never
* that partly relies on an assumption that the system realloc()
* although
* Failure is impossible if newsize <= self.allocated on entry,
* The number of allocated elements may grow, shrink, or stay the
* ob_size to newsize.  If newsize > ob_size on entry, the content
* items[where] = v;
```
static int
ins1(PyListObject *self, Py_ssize_t where, PyObject *v)
{
    Py_ssize_t i, n = Py_SIZE(self);
    PyObject **items;
    if (v == NULL) {
        PyErr_BadInternalCall();
        return -1;
    }
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
                        "cannot add more objects to list");
        return -1;
    }
    if (list_resize(self, n+1) == -1)
        return -1;
    if (where < 0) {
        where += n;
        if (where < 0)
            where = 0;
    }
    if (where > n)
        where = n;
    items = self->ob_item;
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    Py_INCREF(v);
    items[where] = v;
    return 0;
}

int
PyList_Insert(PyObject *op, Py_ssize_t where, PyObject *newitem)
{
    if (!PyList_Check(op)) {
        PyErr_BadInternalCall();
        return -1;
    }
    return ins1((PyListObject *)op, where, newitem);
}

static int
app1(PyListObject *self, PyObject *v)
{
    Py_ssize_t n = PyList_GET_SIZE(self);
    assert (v != NULL);
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
                        "cannot add more objects to list");
        return -1;
    }
    if (list_resize(self, n+1) == -1)
        return -1;
    Py_INCREF(v);
    PyList_SET_ITEM(self, n, v);
    return 0;
}

int
PyList_Append(PyObject *op, PyObject *newitem)
{
    if (PyList_Check(op) && (newitem != NULL))
        return app1((PyListObject *)op, newitem);
    PyErr_BadInternalCall();
    return -1;
}

/* Ensure ob_item has room for at least newsize elements, and set
 * ob_size to newsize.  If newsize > ob_size on entry, the content
 * of the new slots at exit is undefined heap trash; it's the
 * caller's responsibility to overwrite them with sane values.
 * The number of allocated elements may grow, shrink, or stay the
 * same.
 * Failure is impossible if newsize <= self.allocated on entry,
 * although
 * that partly relies on an assumption that the system realloc()
 * never
 * fails when passed a number of bytes <= the number of bytes last
 * allocated (the C standard doesn't guarantee this, but it's hard
 * to
 * imagine a realloc implementation where it wouldn't be true).
 * Note that self->ob_item may change, and even if newsize is less
 * than ob_size on entry.
 */
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated;
    Py_ssize_t allocated = self->allocated;
    /* Bypass realloc() when a previous overallocation is large
     * enough
     * to accommodate the newsize.  If the newsize falls lower than
     * half
     * the allocated size, then proceed with the realloc() to shrink
     * the list.
     */
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
    }
    /* This over-allocates proportional to the list size, making room
     * for additional growth.  The over-allocation is mild, but is
     * enough to give linear-time amortized behavior over a long
     * sequence of appends() in the presence of a poorly-performing
     * system realloc().
     * The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72,
     * 88, ...
     */
    new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
    /* check for integer overflow */
    if (new_allocated > PY_SIZE_MAX - newsize) {
        PyErr_NoMemory();
        return -1;
    } else {
        new_allocated += newsize;
    }
    if (newsize == 0)
        new_allocated = 0;
    items = self->ob_item;
    if (new_allocated <= (PY_SIZE_MAX /
                         sizeof(PyObject *)))
        PyMem_RESIZE(items, PyObject *, new_allocated);
    else
        items = NULL;
    if (items == NULL) {
        PyErr_NoMemory();
        return -1;
    }
    self->ob_item = items;
    Py_SIZE(self) = newsize;
    self->allocated = new_allocated;
    return 0;
}
Py_ssize_t n = PyList_GET_SIZE(self);
assert (v != NULL);
if (n == PY_SSIZE_T_MAX) {
    PyErr_SetString(PyExc_OverflowError, "cannot add more objects to list");
    return -1;
}
if (list_resize(self, n+1) == -1)
    return -1;
Py_INCREF(v);
PyList_SET_ITEM(self, n, v);
return 0;

int PyList_Append(PyObject *op, PyObject *newitem)
{
    if (PyList_Check(op) && (newitem != NULL))
        return app1((PyListObject *)op, newitem);
    PyErr_BadInternalCall();
    return -1;
}

/* Ensure ob_item has room for at least newsize elements, and set
 * ob_size to newsize. If newsize > ob_size on entry, the content
 * of the new slots at exit is undefined heap trash; it's the
caller's
 * responsibility to overwrite them with sane values.
 * The number of allocated elements may grow, shrink, or stay the
 * same.
 * Failure is impossible if newsize <= self.allocated on entry,
 * although
 * that partly relies on an assumption that the system realloc()
 * never
 * fails when passed a number of bytes <= the number of bytes last
 * allocated (the C standard doesn't guarantee this, but it's hard to
 * imagine a realloc implementation where it wouldn't be true).
 * Note that self->ob_item may change, and even if newsize is less
 * than ob_size on entry.
 */
static int list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated;
    Py_ssize_t allocated = self->allocated;
    /* Bypass realloc() when a previous overallocation is large
     * enough
     * to accommodate the newsize. If the newsize falls lower than
     * half
     * the allocated size, then proceed with the realloc() to shrink
     * the list.
     */
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SIZE(self) = newsize;
        return 0;
    }
    /* This over-allocates proportional to the list size, making room
     * for additional growth. The over-allocation is mild, but is
     * enough to give linear-time amortized behavior over a long
     * sequence of appends() in the presence of a poorly-performing
     * system realloc().
     * The growth pattern is:  0, 4, 8, 16, 25, 35, 46, 58, 72,
     * 88, ...
     */
    new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
    /* check for integer overflow */
    if (new_allocated > PY_SIZE_MAX - newsize) {
        PyErr_NoMemory();
        return -1;
    } else {
        new_allocated += newsize;
    }
    if (newsize == 0)
        new_allocated = 0;
    items = self->ob_item;
    if (new_allocated <= (PY_SIZE_MAX / sizeof(PyObject *)))
        PyMem_RESIZE(items, PyObject *, new_allocated);
    else
        items = NULL;
    if (items == NULL) {
        PyErr_NoMemory();
        return -1;
    }
    self->ob_item = items;
    Py_SIZE(self) = newsize;
    self->allocated = new_allocated;
    return 0;
}
Py_ssize_t allocated = self->allocated;

/* Bypass realloc() when a previous overallocation is large enough to accommodate the newsize. If the newsize falls lower than half the allocated size, then proceed with the realloc() to shrink the list. */
if (allocated >= newsize && newsize >= (allocated >> 1)) {
    assert(self->ob_item != NULL || newsize == 0);
    Py_SIZE(self) = newsize;
    return 0;
}

/* This over-allocates proportional to the list size, making room for additional growth. The over-allocation is mild, but is enough to give linear-time amortized behavior over a long sequence of appends() in the presence of a poorly-performing system realloc(). The growth pattern is: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...
 */
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);

/* check for integer overflow */
if (new_allocated > PY_SIZE_MAX - newsize) {
    PyErr_NoMemory();
    return -1;
} else {
    new_allocated += newsize;
}

if (newsize == 0)
    new_allocated = 0;
items = self->ob_item;
if (new_allocated <= (PY_SIZE_MAX / sizeof(PyObject *)))
    PyMem_RESIZE(items, PyObject *, new_allocated);
else
    items = NULL;
if (items == NULL) {
    PyErr_NoMemory();
    return -1;
}
self->ob_item = items;
Py_SIZE(self) = newsize;
self->allocated = new_allocated;
return 0;
Python doesn’t have a true low-level array, but we can pretend the List is an array if we limit ourselves to using:

- [] for index-based access — assume this is $O(1)$
- len() for determining the size of a list
- l.append(x) to increase the size of a list by 1 elements — use this sparingly, as it is $\neq O(1)$!
- … all other operations (e.g., “in”, “not in”, min/max, count, insert, “del”, pop, remove, reverse) are off-limits!

Our list with a prepend operation:

```python
class MyList(list):
    def prepend(self, val):
        self.append(0)  # grow the list by one element
        for i in range(len(self)-1, 0, -1):
            self[i] = self[i-1]
        self[0] = val
```

• Upcoming MP4: implement an array-based List ADT!