COMP 6411: Comparative Programming Languages

Tutorial 1: Python Intro

Topics

- Welcome to Python
- Guiding principles of Python design
- Python data types
  - Ints and floats
  - Strings
  - Lists
- A brief look at computational complexity

Origins

- Python is a high-level, interpreted, general purpose programming language.
- Development began in the late 1980s with the first release in 1991.
- The original author was Guido van Rossum
  - He remains the principal author/controller
  - Benevolent Dictator for Life (BDFL)
- That original version was written in a very short time (days/weeks)
  - However, Python is now a large language with a very extensive set of libraries

The Guiding Principles

- Before looking at the structure of the language, it is useful to understand the guiding principles of Python
- In other words, what are the language creators trying to do with Python.
- This has been formally codified:
  - The Zen of Python
  - A collection of software principles supported by the language

The Zen of Python

- Principles 1 to 10:

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Zen…cont'd

- Principles 11 to 20:

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one—and preferably only one—obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than right now.[5]

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea—let's do more of those!

Zen summary

- Python is a language that is meant to be simple and very readable.
- It avoids the complexity often used to support obscure problems.
- It's syntax is largely self-documenting
- Most common tasks have one (or a small number) way to be implemented
  - So Python typically offers one good/main way to do something (e.g., a FOR loop)
- Minimal "mysterious" things happening transparently.
- Summary: As much as possible, Python supports good software engineering practices

So what is Python

- We will be looking at Python syntax and structure in the slides that follow.
- But, as a quick look ahead, we can expect to see the following:
  1. An imperative language with syntax roughly similar to Java/C
  2. Extremely readable code
     - Uses white space for indentation!!! (no curly braces)
  3. Object Orientation (non OOP also possible)
  4. Automatic memory management (i.e., garbage collection)
  5. No static typing
     - Suitability of actions/methods is determined at run-time

Python Basics

- Before creating useful python programs, we we will first look at some basic syntax.
- While Python uses fairly recognizable syntax (assignment statements, WHILE loops, etc.), it is not a true C-style language
- In other words, it has some syntax that you simply do not see in C/C++/Java.
- So we will try to highlight some of the obvious differences.

Python Versions

- For many years, the standard line of Python was 2.x
- Python 3 was released in 2008
  - It changes various elements of the API and is NOT backwards compatible with 2.x
  - …though many version 3 updates have been back-ported to version 2.
- Today, the most common Python versions are probably 2.7 and 3.3
  - 2.7's "end of life" is set at 2020.
- For an intro to Python, there isn't much of a difference
  - However, you will note that Python 3.x treats `print` as a function, rather than a statement
    - i.e., `print ("hello")` rather than `print "hello"`)

Using the command line

- Python applications are written like programs in any other language
  - We create a set of source files that are combined in some way into a final application
  - Typically we use an IDE for the development process
- However, it is also possible to run the Python interpreter interactively (i.e., from the command line).
- This can be especially useful as you explore the syntax of Python, particularly basic data types and functions
  - To use the interpreter this way, simply open a terminal and type “python” at the command line.
  - Use CTRL-D or quit() to exit the interpreter.

Basic data types

- Python provides support for the standard integer and floating point values
- However, Python does not require explicit type definitions, like `int`, `float`, and `double`.
- So assignments can be done simply as:

```
foo = 26
bar = 768.56
```

- Note that there are no “type keywords” and no “;” to end each statement

Numeric implementation

- Python determines the internal storage format for numbers automatically.
- For example, in most cases, integer values are stored internally in a C language `signed long int`.
- In contrast, floating point values are stored in a C `double`.
- It is also possible to provide an "L" suffix after an integer value.
  - `34534334634634L`
- There is no limit on the size of such long integers (other than the size of memory).
  - Internally, this will be stored as an array of digits
  - Processing of Python long integers is slower since software libraries must manipulate these "digit arrays"

Strings

- You will be happy to know that string processing in Python is VERY intuitive
- Strings in Python are enclosed either in single quotes (‘foo’) or double quotes (“foo”).
  - There is no difference in meaning
- Escape characters like ‘\n’ can be included as well
  - Note that invalid escape characters will just be treated as regular text

Raw Strings

- You can also “turn off” escape characters by using *raw* strings
  - We preface the string with ‘r’

```
myPath = “C:\foo\noo”  #note the \n
>> C:\foo
>> oo

myRawPath = r“C:\foo\noo”
>> C:\foo\noo
```

Multi-line strings

- It is possible to enclose strings in a triple quote (using either single quotes or double quotes)
- This will concatenate all lines into a single string

```
myNote = """
   Roses are red
   Violets are blue
   yada yada yada
"""
```

- The interpreter will automatically add the new line characters
  - To prevent this, you can add a "\" at the end of each line

String concatenation

- Strings can be concatenated with the "+" operator.

```
s1 = "big" + "dog" # bigdog
s2 = "big" + s1    # bigbigdog"
```

- You can also repeat string contents with the "*" operator

```
s3 = "big" * 3     # bigbigbig
```

String indexing

- You can also treat a string like an array of characters which, of course, is how it will be stored internally

```
 s1 = "bigdog"
 print s1[1]
>> i
```

- Python provides bounds checking for strings
- We can also use negative array indexes
  - [-1] is the last char, [-2] the second last, etc.

```
 s1 = "bigdog"
 print s1[-2] # last char is [-1]
>> o
```

String slicing

- The indexing concept is extended to support the notion of string slices

```
s1 = “bigdog”
s1[2:4]  # gd
s1[2:]   # gdog
s1[:4]   # bigd
s1[-2:]  # og
s1[2:-2] # gd
s1[2:2]  #
s1[2:44] # gdog    note that this works
```

- **Important**: Slices use inclusive/exclusive format
  - In other words, the start index is inclusive, while the end index is exclusive
  - So x[2:4] means x[2] to x[3]

String immutability

- As is the case with Java, Python strings are immutable.
  - So this does NOT work

```
s1 = “bigdog”
s1[2] = “G”
```

- However, you can accomplish the same assignment by creating a new string

```
s1 = “bigdog”
s2 = s1[:2] + “G” + s1[3:]
```

Composite data types

- Python includes a number of built-in composite data types
  - Lists
  - Tuples
  - Sequences
  - Sets
  - Dictionaries
- Of these, lists may be the most common, and the most versatile, so we will take a look at these first

Lists

- Lists are written as a comma separated set of values, enclosed in square brackets

```
list1 = [4, 6, 7, 99]
```

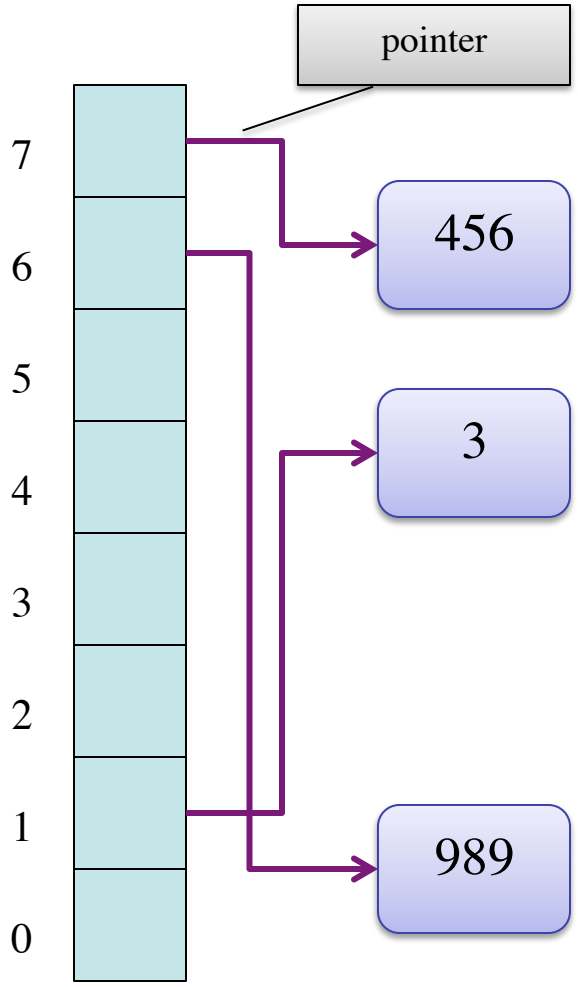
- As is the case with strings, we can use indexing syntax and list slicing

```
list1[1]    # 6
list1[2:]   # [7, 99]
```

- Slice operations on a list return a new list

List implementation

- Internally, lists are not likely to be implemented as actual linked lists.
  - i.e., list "nodes" linked together with pointers
- Performance on linked lists can be quite poor, particularly for indexing operations.
- In CPython, the most common version of the Python interpreter, the list is built using a dynamically resized array.
  - Here, each cell in the array holds a pointer to the object that is associated with that index



Mixed lists

- Lists can be constructed with great flexibility
- For example, it is possible to mix types within a list
  - There is no notion of “generics”, as with Java

```
list2 = [4, 93, “bigdog”, 2]
```

- You can also *nest* lists

```
list3 = [“foo”, “boo”]
list4 = [1, 2, 8, list3, [12, 9]]
```

Mutability

- Unlike strings, Python lists are mutable.
- So you can change the contents of a list without creating a new list.

```
list3 = [“foo”, “boo”]
list4 = [1, 2, 8, list3, [12, 9]]
list4[4] = 16
print list4  # replaced [12, 9]
>>> [1, 2, 8, [“foo”, “boo”], 16]

list4[1:3] = [“do”, “not”]
print list4  # replaced [2, 8]
>>> [1, “do”, “not”, [“foo”, “boo”], 16]
```

List methods

- Lists also support a group of methods that provide advanced functionality.
- These methods are invoked using the standard OOP-style "." notation.
- List methods include (where "list" is the name of the list variable):
  - `list.append(x)`: add x to the end of the list
  - `list.extend(L)`: append list L to current list
  - `list.insert(i, x)`: insert item x at position i
  - `list.remove(x)`: remove first occurrence of x

List methods…cont'd

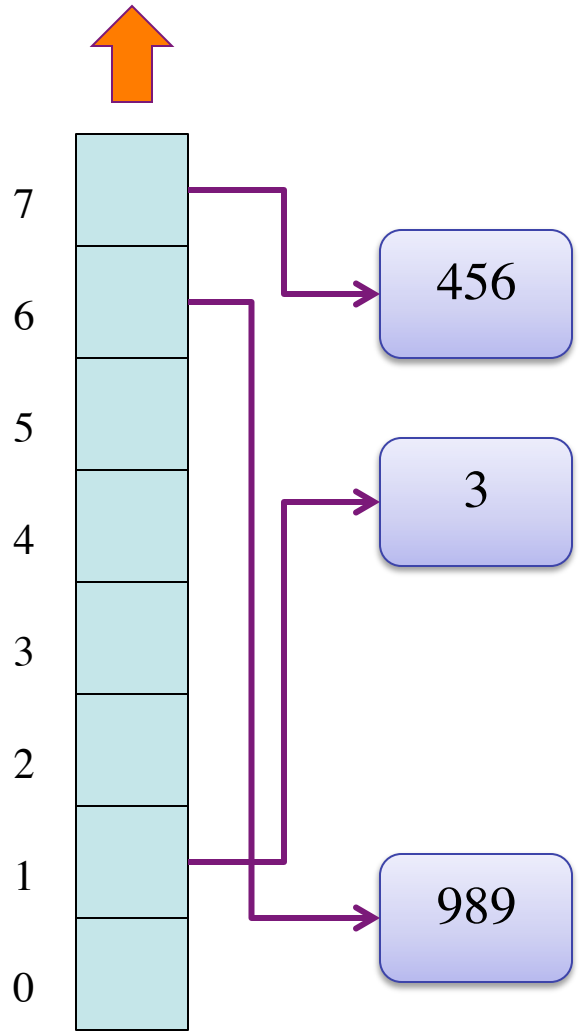
- Additional methods include:
  - `list.pop()`: remove and return the last element
  - `list.pop(i)`: pop element at position i
  - `list.index(x)`: return first index position of x
  - `list.count(x)`: number of instances of x
  - `list.reverse()`: reverse the order of elements
  - `list.sort()`: sort elements "in place" (optional args are possible with the sort method in order to provide customized sorting)

Computational Complexity

- It is often useful to be aware of the underlying cost of various operations.
- While all of the previous methods will give correct answers, the run-time in practice can sometimes surprise you.
- For example, let's assume that we are processing a large list that contains 1 million elements.
- Further, we are going to perform one of the following three operations on it.
  1. list[6] = 43
  2. list.append(43)
  3. list.insert(6, 43)
- All operations will work just fine, and modify the list with a single update.
  - But is there any performance difference in practice?

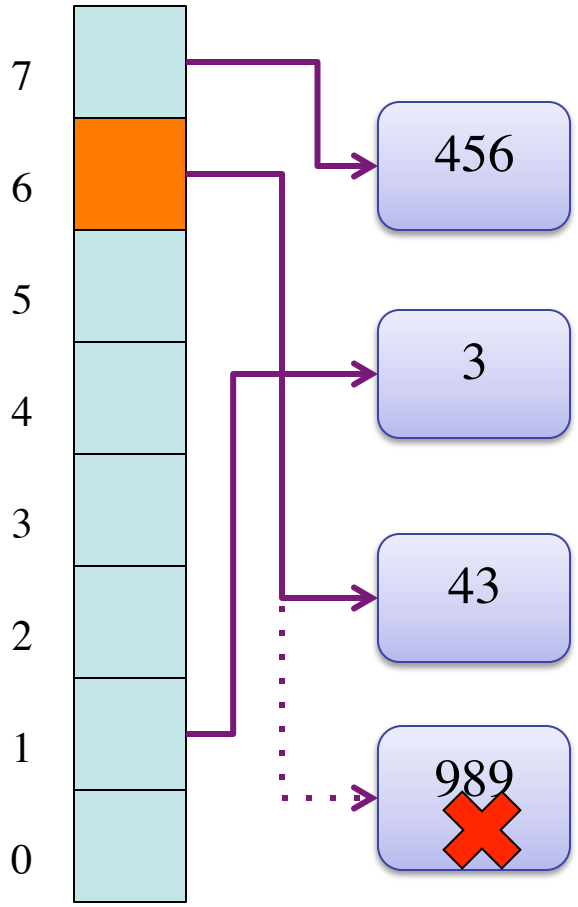
Before we begin…

- Recall that the Python interpreter is written in another language, typically C.
- So a list is generally implemented internally as an array of pointers
- As a result, when we consider operation cost, we must think in terms of this structure.



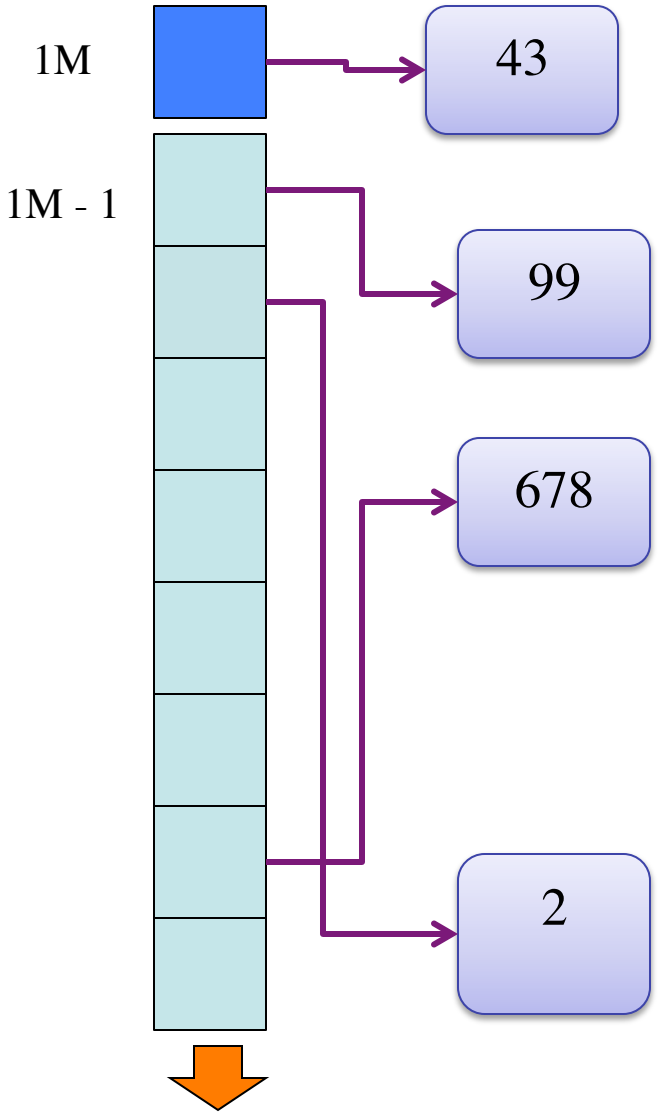
Operation 1: update by subscript

- Here, we are modifying a single element in the list.
- This is what is known as an O(1) or *constant time* operation
- In other words, there is a constant number of steps required to do this update
- In practice, this update is virtually instantaneous
- Moreover, it does **NOT** matter which subscript we use.



Operation 2: append

- Here, we are modifying the list by adding an element to the end of the array
- What is the cost?
- In a purely theoretical sense, it is also O(1).
- After all, we simply create a new element and assign it's pointer to position 1M.
- This can be done in a fixed number of steps

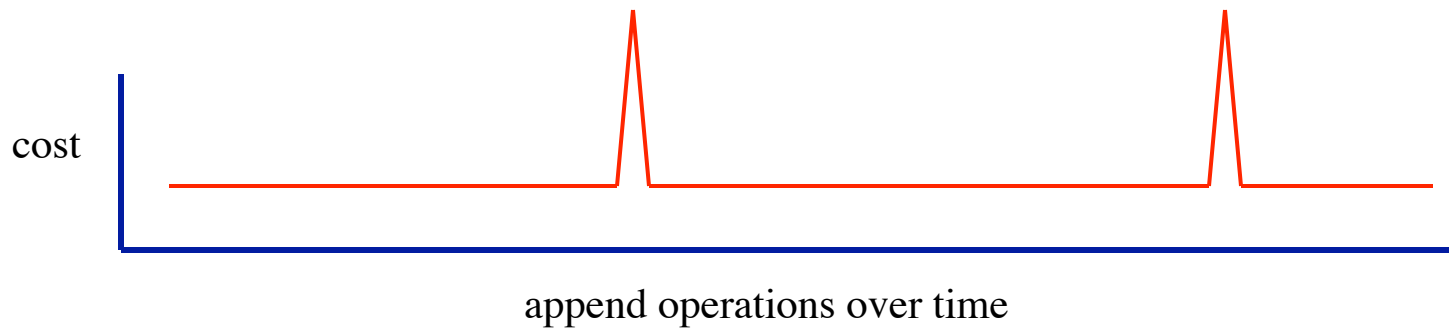


Operation 2…cont'd

- In a practical sense, however, it's a little more complicated
- Appending an element WHEN THERE IS SPACE is indeed O(1).
- What happens when the array is full?
- In short, we have to use something like C's `realloc` function.
  - Create a new larger array and copy all of the previous elements to that new space.

Operation 2…cont'd

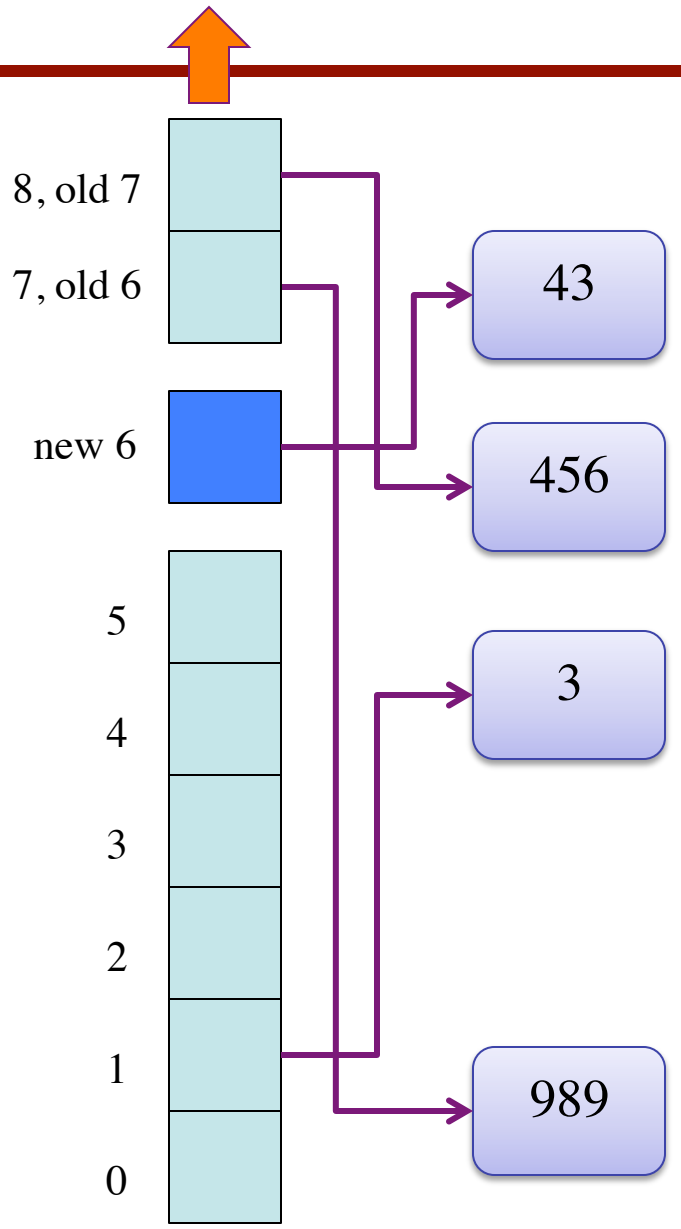
- This creates a pattern of a large number of O(1) append operations, followed by occasional O(n), or linear time, steps
  - O(n) implies that the cost is proportional to the number of elements *n*.
- So graphically, the performance cost might look like this:



- The point is that a small number of appends may appear to be much more expensive that others

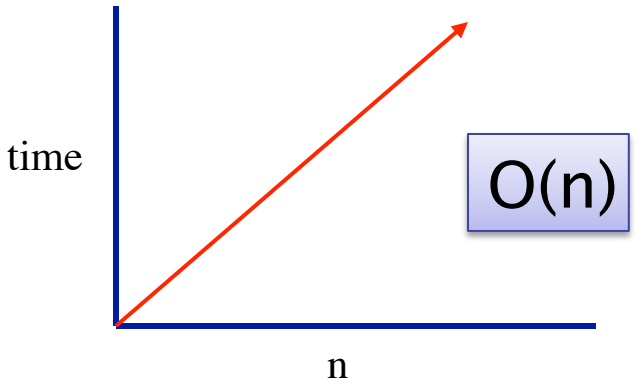
Operation 3: insert

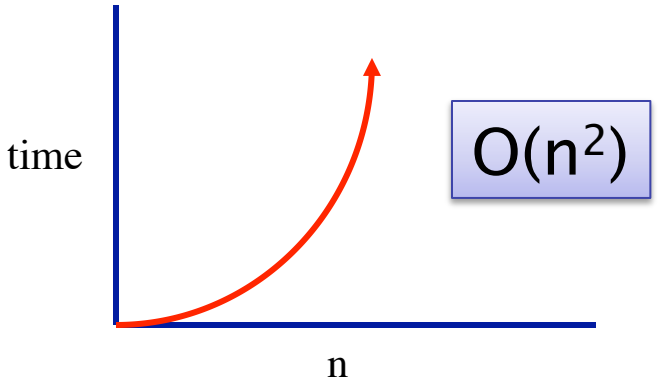
- In this case, we must insert an item into an existing list.
- So what is the cost, under the assumption that the underlying data structure is an array
- In this case, we must shift every element in the array (after the insertion point)
- On average this is O(n) on every insertion!
  - Note that the "average" is (1/2)*n, but we discard constant values like ½ in this kind of analysis.



Operation 3 cont'd

- So now our performance curve looks MUCH different
  - Time is now directly proportional to n
- What if we build our array by inserting an item at the head of the list each time?
- Now, for each of the n items we insert, we must make O(n) shifts to make room for it.
  - This leads to an $O(n^2)$ cost for the whole operation.
- Using "real" numbers, for 1 million items the number of operations would be on the order of 1 trillion!





Summary of complexity

- This doesn't mean that you should not use these methods.
  - They are very useful and work very well in most cases.
- Instead, the point is to understand how various data structures may be represented internally and how this might affect performance.
  - On small "toy" applications it doesn't matter but in "big" applications it might.
- When you write, test, and profile your code, try to be aware of data representation issues.
  - So when you see performance degrade quickly, for no apparent reason, it may be related to the underlying computational complexity.