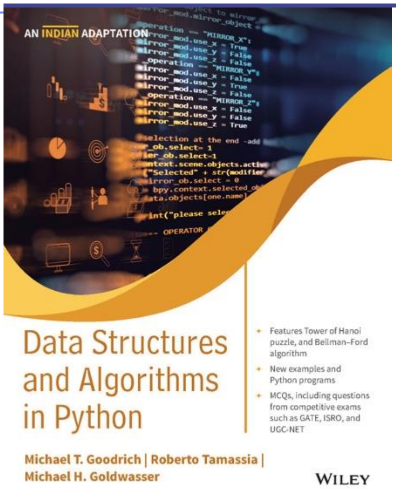


Data Structures and Algorithms in Python

Michael T. Goodrich/ Roberto Tamassia/ Michael H. Goldwasser

Adapted by Subhrakanta Panda

Chapter 2 and 5

Object Oriented Programming.
Array-Based Sequences

Class Definitions

- A class serves as the primary means for abstraction in object-oriented programming.
- In Python, every piece of data is represented as an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

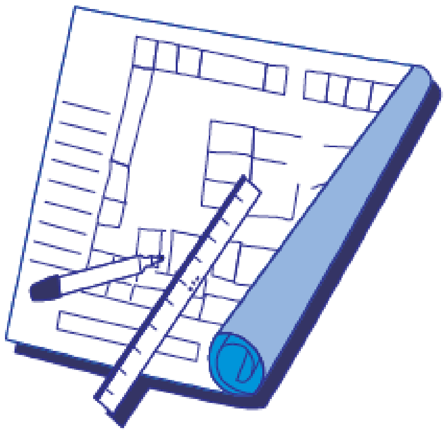
Terminology

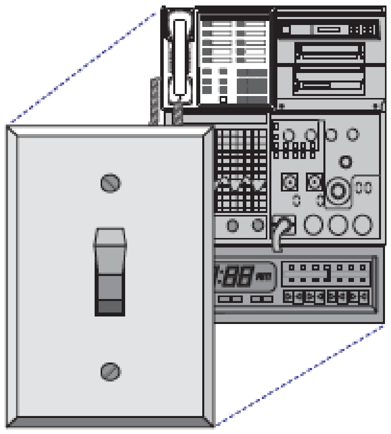
- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**, also known as **data members**, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

Object-Oriented Design Principles

- Modularity
- Abstraction
- Encapsulation

Modularity


Abstraction


Encapsulation

The **self** Identifier

- In Python, the **self** identifier plays a key role.
- In any class, there can possibly be many different instances, and each must maintain its own instance variables.
- Therefore, each instance stores its own instance variables to reflect its current state. Syntactically, **self** identifies the instance upon which a method is invoked.

Example

```
class CreditCard:
  """A consumer credit card."""

  def __init__(self, customer, bank, acnt, limit):
    """Create a new credit card instance.

    The initial balance is zero.

    customer  the name of the customer (e.g., 'John Bowman')
    bank      the name of the bank (e.g., 'California Savings')
    acnt      the acount identifier (e.g., '5391 0375 9387 5309')
    limit     credit limit (measured in dollars)
    """
    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0

```

Example, Part 2

```
  def get_customer(self):
    """Return name of the customer."""
    return self._customer

  def get_bank(self):
    """Return the bank's name."""
    return self._bank

  def get_account(self):
    """Return the card identifying number (typically stored as a string)."""
    return self._account

  def get_limit(self):
    """Return current credit limit."""
    return self._limit

  def get_balance(self):
    """Return current balance."""
    return self._balance
```

Example, Part 3

```
def charge(self, price):
  """Charge given price to the card, assuming sufficient credit limit.

  Return True if charge was processed; False if charge was denied.
  """
  if price + self._balance > self._limit:      # if charge would exceed limit,
    return False                               # cannot accept charge
  else:
    self._balance += price
    return True

def make_payment(self, amount):
  """Process customer payment that reduces balance."""
  self._balance −= amount
```

Constructors

- A user can create an instance of the CreditCard class using a syntax as:

```
cc = CreditCard('John Doe, '1st Bank', '5391 0375 9387 5309', 1000)
```

- Internally, this results in a call to the specially named `__init__` method that serves as the constructor of the class.
- Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables.

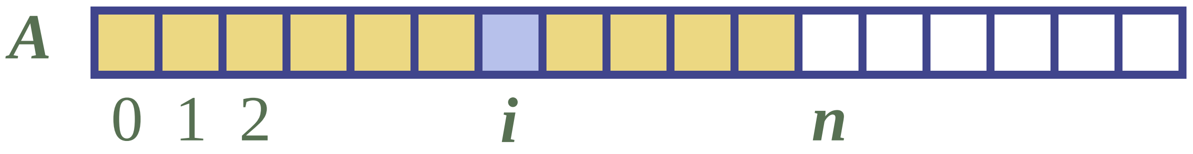
Array-Based Sequences

Abstract Data Types

- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- An ADT specifies what each operation does, but not how it does it.
- The collective set of behaviors supported by an ADT is its **public interface**.

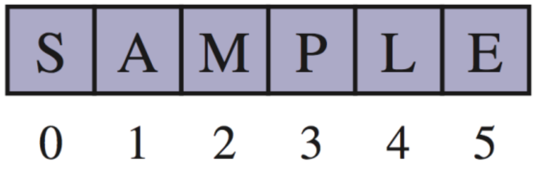
Python Sequence Classes

- Python has built-in types, **list**, **tuple**, and **str**.
- Each of these **sequence** types supports indexing to access an individual element of a sequence, using a syntax such as A[i]
- Each of these types uses an **array** to represent the sequence.
  - An array is a set of memory locations that can be addressed using consecutive indices, which, in Python, start with index 0.

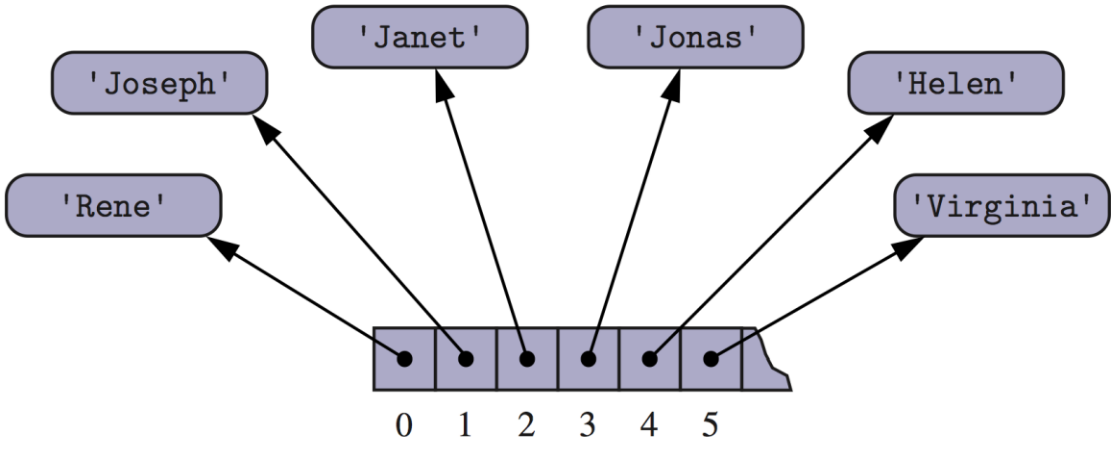


Arrays of Characters or Object References

- An array can store primitive elements, such as characters, giving us a **compact array.**



- An array can also store references to objects.



Compact Arrays

- Primary support for compact arrays is in a module named **array**.
  - That module defines a class, also named array, providing compact storage for arrays of primitive data types.
- The constructor for the array class requires a type code as a first parameter, which is a character that designates the type of data that will be stored in the array.

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

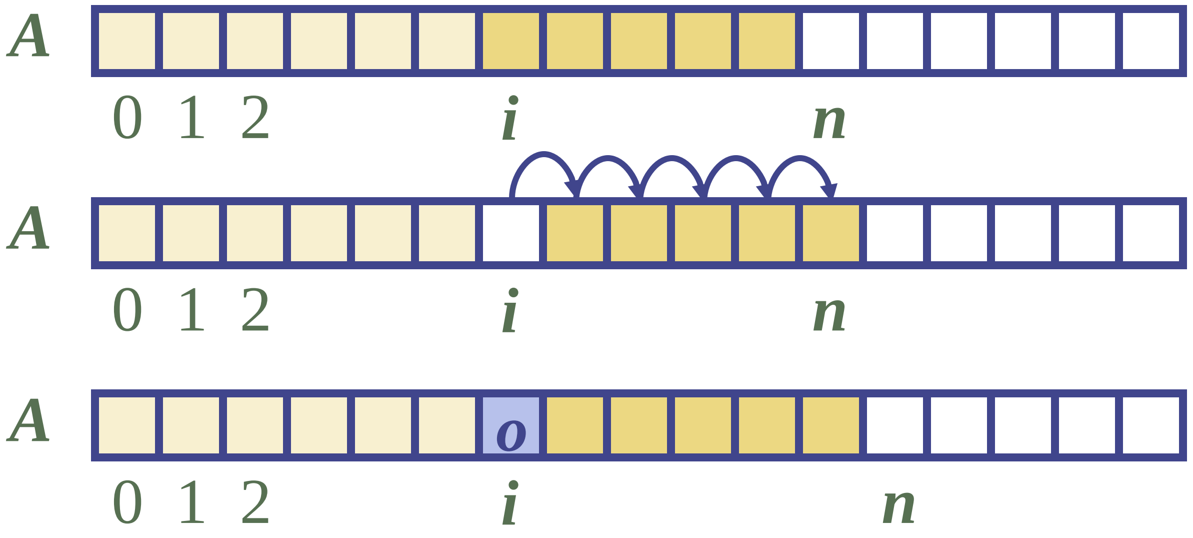
Type Codes in the array Class

- Python's array class has the following type codes:

Code	C Data Type	Typical Number of Bytes
`'b'`	signed char	1
`'B'`	unsigned char	1
`'u'`	Unicode char	2 or 4
`'h'`	signed short int	2
`'H'`	unsigned short int	2
`'i'`	signed int	2 or 4
`'I'`	unsigned int	2 or 4
`'l'`	signed long int	4
`'L'`	unsigned long int	4
`'f'`	float	4
`'d'`	float	8

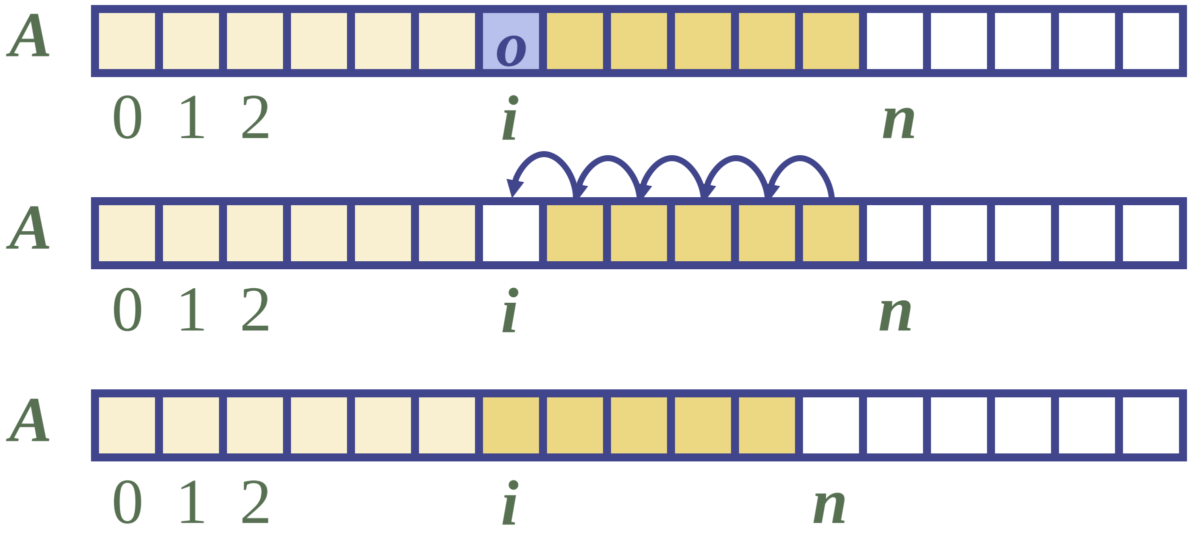
Insertion

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \dots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

- In an array based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at I takes $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time in worst case
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one…

Growable Array-based Array List

- In an add(o) operation (without an index), we could always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

```
Algorithm add(o)
  if $t = S.length - 1$ then
    $A \leftarrow$ new array of size …
    for $i \leftarrow 0$ to $n-1$ do
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

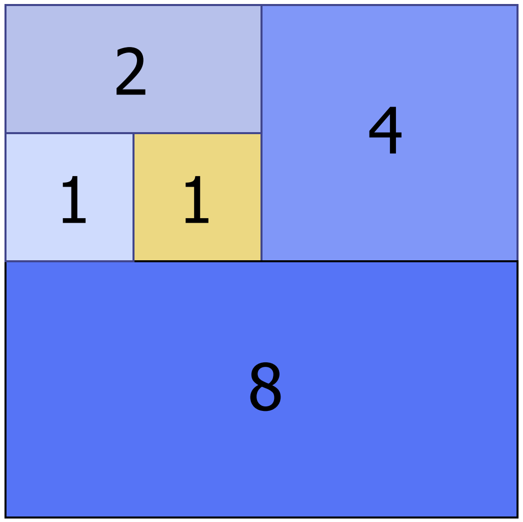
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



Python Implementation

```
import ctypes                                    # provides low-level arrays

class DynamicArray:
  """A dynamic array class akin to a simplified Python list."""

  def __init__(self):
    """Create an empty array."""
    self._n = 0                                  # count actual elements
    self._capacity = 1                           # default array capacity
    self._A = self._make_array(self._capacity)   # low-level array

  def __len__(self):
    """Return number of elements stored in the array."""
    return self._n

  def __getitem__(self, k):
    """Return element at index k."""
    if not 0 <= k < self._n:
      raise IndexError('invalid index')
    return self._A[k]                            # retrieve from array

  def append(self, obj):
    """Add object to end of the array."""
    if self._n == self._capacity:                # not enough room
      self._resize(2 * self._capacity)           # so double capacity
    self._A[self._n] = obj
    self._n += 1

  def _resize(self, c):                          # nonpublic utitity
    """Resize internal array to capacity c."""
    B = self._make_array(c)                      # new (bigger) array
    for k in range(self._n):                     # for each existing value
      B[k] = self._A[k]
    self._A = B                                  # use the bigger array
    self._capacity = c

  def _make_array(self, c):                      # nonpublic utitity
     """Return new array with capacity c."""
     return (c * ctypes.py_object)( )            # see ctypes documentation
```