Structures de données avancées en python

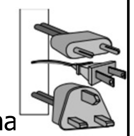
Mariem Gzara
Engineering program
Department of computer science
Higher School of mathematics and computer science
University of Monastir

OBJECT ORIENTED PROGRAMMING

Object-Oriented Design Goals

- **Robustness:** a program produces the right output for all the anticipated inputs in the program's application.

- **Adaptability:** Able to evolve over time in response to changing conditions in its environment (also called evolvability),
  - **portability,** ability of software to run with minimal change on different hardware and operating system platforms.



- **Reusability**: the same code should be usable as a component of different systems in various applications.

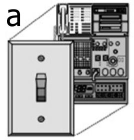
Object-Oriented Design Principles

- **Modularity :** several different components that must interact correctly in order for the entire system to work properly.
  - a module is a collection of closely related functions and classes that are defined together in a single file of source code.

- **Abstraction:** notion of abstraction is to distill a complicated system down to its most fundamental parts.
  - Abstract data types (ADTs).
  - A mathematical 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.

Object-Oriented Design Principles

- **Encapsulation:** Different components of a software system should not reveal the internal details of their respective implementations.



Software Development

- Three major steps are:
  1. Design
  2. Implementation
  3. Testing and Debugging

Class Definitions

- Attributes (also known as fields, instance variables, or data members
- Member functions (also known as methods)

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

```
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

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

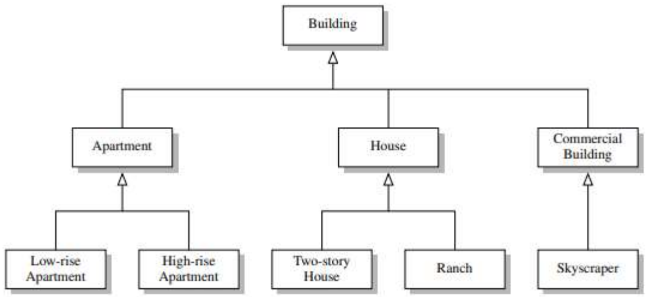
The self Identifier: self identifies the instance upon which a method is invoked

__init__ method that serves as the constructor of the class

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


```
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

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
```


Inheritance

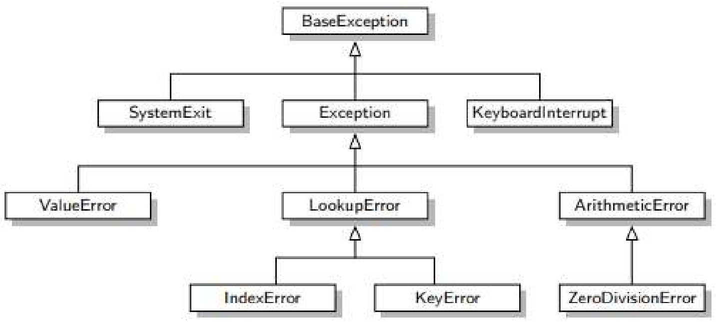


- A house is a building, and a ranch is a house.
- Using mathematical notations, the set of houses is a subset of the set of buildings, but a superset of the set of ranches.

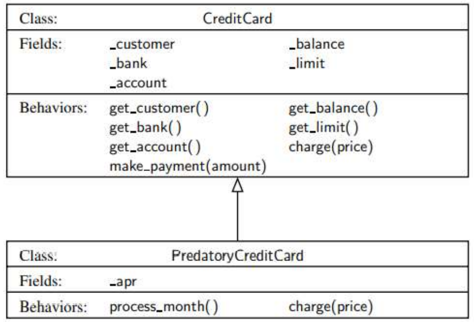
Inheritance

- Organize various structural components of a software package in a hierarchical fashion.
- The correspondence between levels is often referred to as an "is a" relationship
- Base class, parent class, or superclass, while the newly defined class is known as the subclass or child class
- A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method
- A subclass may also extend its superclass by providing brand new methods

Python's Exception Hierarchy



Extending the CreditCard Class



1. If an attempted charge is rejected because it would have exceeded the credit limit, a $5 fee will be charged.
2. there will be a mechanism for assessing a monthly interest charge on the outstanding balance, based upon an Annual Percentage Rate (APR) specified as a constructor parameter


```
class PredatoryCreditCard(CreditCard):
  """An extension to CreditCard that compounds interest and fees."""

  def __init__(self, customer, bank, acnt, limit, apr):
    """Create a new predatory 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)
    apr       annual percentage rate (e.g., 0.0825 for 8.25% APR)
    """
    super().__init__(customer, bank, acnt, limit)    # call super constructor
    self._apr = apr
```



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

  Return True if charge was processed.
  Return False and assess $5 fee if charge is denied.
  """
  success = super().charge(price)          # call inherited method
  if not success:
    self._balance += 5                     # assess penalty
  return success                           # caller expects return value

def process_month(self):
  """Assess monthly interest on outstanding balance."""
  if self._balance > 0:
    # if positive balance, convert APR to monthly multiplicative factor
    monthly_factor = pow(1 + self._apr, 1/12)
    self._balance *= monthly_factor
```

Abstract Base Classes

- An abstract base class is one that cannot be directly instantiated, while a concrete class is one that can be instantiated
- In classic object-oriented terminology, we say a class is an abstract base class if its only purpose is to serve as a base class through inheritance

Abstract Classes

- Considered as a blueprint or template for other classes.
- Contains one or more abstract methods.
- An abstract method is a method that has declaration but no implementation.
- Abstract classes can not be instantiated. It needs subclasses (child classes) to provide implementation.
- Abstract classes are required for providing “Abstraction” or a simplified interface (API) while hiding the underlying implementation.
- Python provides abstract classes by declaring abstract base class (ABC) which could be inherited by other child classes.


```python
from abc import ABCMeta, abstractmethod          # need these definitions

class Sequence(metaclass=ABCMeta):
  """Our own version of collections.Sequence abstract base class."""

  @abstractmethod
  def __len__(self):
    """Return the length of the sequence."""

  @abstractmethod
  def __getitem__(self, j):
    """Return the element at index j of the sequence."""

  def __contains__(self, val):
    """Return True if val found in the sequence; False otherwise."""
    for j in range(len(self)):
      if self[j] == val:                           # found match
        return True
    return False

```



```python

  def index(self, val):
    """Return leftmost index at which val is found (or raise ValueError)."""
    for j in range(len(self)):
      if self[j] == val:                               # leftmost match
        return j
    raise ValueError('value not in sequence')          # never found a match

  def count(self, val):
    """Return the number of elements equal to given value."""
    k = 0
    for j in range(len(self)):
      if self[j] == val:                               # found a match
        k += 1
    return k
```

Nested Classes

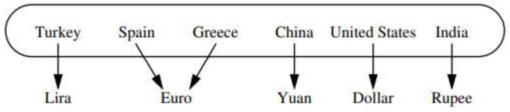
- Nest one class definition within the scope of another class
  - class A: # the outer class class
  - B: # the nested class
  - Class B is the nested class.
  - The identifier B is entered into the namespace of class A associated with the newly defined class.
- This technique is unrelated to the concept of inheritance, as class B does not inherit from class A ...

21

21

Maps and Dictionaries

- Python's dict class is arguably the most significant data structure in the language.
- It represents an abstraction known as a dictionary in which unique keys are mapped to associated values
- Dictionaries: arrays or maps



A map from countries (the keys) to their units of currency (the values).

22

22

The Map ADT

- M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError. (python: __getitem__).
- M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. (Python: __setitem__)
- del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError. (Python: __delitem__).
- len(M): Return the number of items in map M. (Python: __len__)
- iter(M): The default iteration for a map generates a sequence of keys in the map.(Python: __iter__)

23

23

The Map ADT

- k in M: Return True if the map contains an item with key k. (Python: special contains method.)
- M.get(k, d=None): Return M[k] if key k exists in the map; otherwise return default value d. This provides a form to query M[k] without risk of a KeyError.
- M.setdefault(k, d): If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.
- M.pop(k, d=None): Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d (or raise KeyError if parameter d is None)

24

24

The Map ADT

- M.popitem( ): Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.
- M.clear( ): Remove all key-value pairs from the map.
- M.keys( ): Return a set-like view of all keys of M.
- M.values( ): Return a set-like view of all values of M.
- M.items( ): Return a set-like view of (k,v) tuples for all entries of M.
- M.update(M2): Assign M[k] = v for every (k,v) pair in map M2.
- M == M2: Return True if maps M and M2 have identical key-value associations.
- M != M2: Return True if maps M and M2 do not have identical keyvalue associations.

The Map ADT

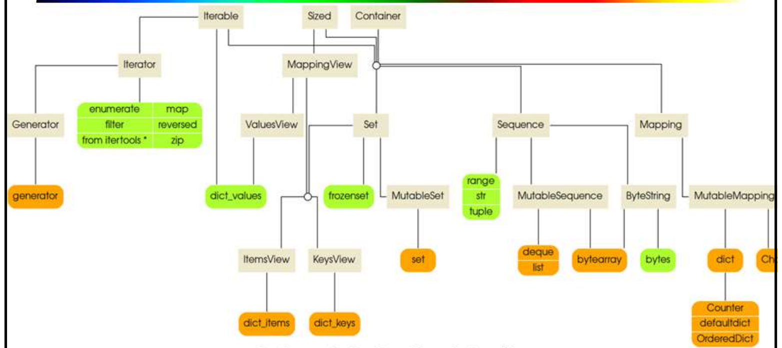
Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys( )	'B', 'U'	{'B': 4, 'U': 2}
M.values( )	4, 2	{'B': 4, 'U': 2}
M.items( )	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem( )	('B', 4)	{'A': 1, 'U': 2}

We use the literal syntax for Python's dict class to describe the map contents.

Application: Counting Word Frequencies

```
freq = { }
for piece in open(filename).read( ).lower( ).split( ):
  # only consider alphabetic characters within this piece
  word = ''.join(c for c in piece if c.isalpha( ))
  if word:    # require at least one alphabetic character
    freq[word] = 1 + freq.get(word, 0)

max_word = ''
max_count = 0
for (w,c) in freq.items( ):          # (key, value) tuples represent (word, count)
  if c > max_count:
    max_word = w
    max_count = c
print('The most frequent word is', max_word)
print('Its number of occurrences is', max_count)
```


Python Containers



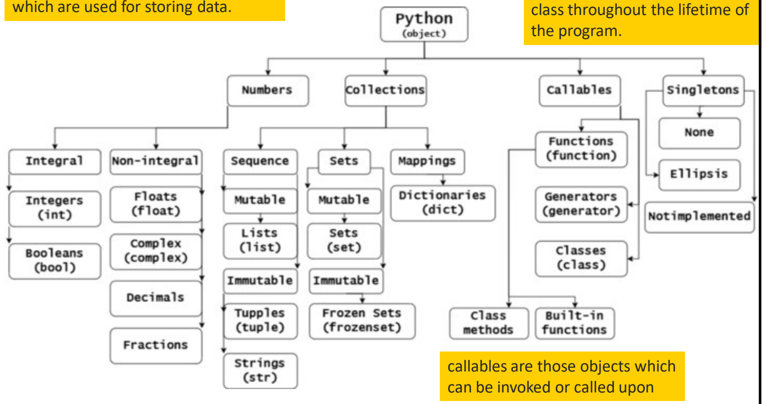
Python 3 Collection Inheritance

Inheritance relations of collections in Python 3.
orange - mutable
green - hashable/immutable
grey - abstract base class from collections.abc

Python Containers

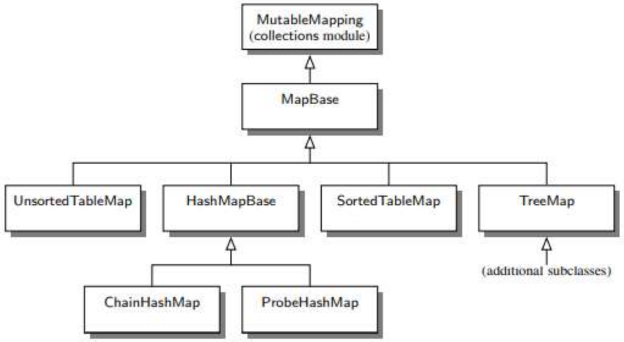
Collections are containers in Python which are used for storing data.

A singleton is a design pattern in Python that allows you to create just one instance of a class throughout the lifetime of the program.



callables are those objects which can be invoked or called upon

Python's MutableMapping Abstract Base Class



MapBase

```
class MapBase(MutableMapping):
  """Our own abstract base class that includes a nonpublic _Item class."""

  #------------------------------- nested _Item class -------------------------------
  class _Item:
    """Lightweight composite to store key-value pairs as map items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __eq__(self, other):
      return self._key == other._key     # compare items based on their keys

    def __ne__(self, other):
      return not (self == other)         # opposite of __eq__

    def __lt__(self, other):
      return self._key < other._key      # compare items based on their keys
```


HashMapBase

```
class HashMapBase(MapBase):
  """Abstract base class for map using hash-table with MAD compression."""

  def __init__(self, cap=11, p=109345121):
    """Create an empty hash-table map."""
    self._table = cap * [ None ]
    self._n = 0                                # number of entries in the map
    self._prime = p                            # prime for MAD compression
    self._scale = 1 + randrange(p-1)           # scale from 1 to p-1 for MAD
    self._shift = randrange(p)                 # shift from 0 to p-1 for MAD

  def _hash_function(self, k):
    return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)
```

HashMapBase

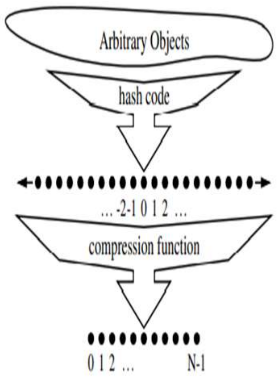
```
def __len__(self):
  return self._n

def __getitem__(self, k):
  j = self._hash_function(k)
  return self._bucket_getitem(j, k)          # may raise KeyError

def __setitem__(self, k, v):
  j = self._hash_function(k)
  self._bucket_setitem(j, k, v)              # subroutine maintains self._n
  if self._n > len(self._table) // 2:        # keep load factor <= 0.5
    self._resize(2 * len(self._table) - 1)   # number 2^x - 1 is often prime
```


HashMapBase

```
def __delitem__(self, k):
  j = self._hash_function(k)
  self._bucket_delitem(j, k)                 # may raise KeyError
  self._n -= 1

def _resize(self, c):                        # resize bucket array to capacity c
  old = list(self.items())                   # use iteration to record existing items
  self._table = c * [None]                   # then reset table to desired capacity
  self._n = 0                                # n recomputed during subsequent adds
  for (k,v) in old:
    self[k] = v                              # reinsert old key-value pair
```


Hash Functions

- Hash function, h, maps each key k to an integer in the range [0,N − 1], where N is the capacity of the bucket array for a hash table.
- "A hash function, h(k), as consisting of two portions:
  - A hash code that maps a key k to an integer,
  - and a compression function that maps the hash code to an integer within a range of indices, [0,N −1], for a bucket array



Hash Codes

- Treating the Bit Representation as an Integer:
  Integer 32 bits.
  (-) 64 bits?

2. Any object x whose binary representation can be viewed as an n-tuple (x0,x1,...,xn−1) of 32-bit integers:
   $\sum_{i=0}^{n-1} x_i$ as x0⊕x1⊕⋯⊕xn−1, where the ⊕ symbol represents the bitwise exclusive-or operation (^ in Python)
   (-)collisions for common groups of strings :"temp01" and "temp10" collide using this function ("stop", "tops", "pots", and "spot").

3. Polynomial Hash Codes : take into consideration the positions of the xi's. choose a nonzero constant, a ≠ 1
   $$x_0 a^{n-1} + x_0 a^{n-2} + \cdots + x_{n-2} a \;\; + x_{n-1}$$
   By Horner's rule :
   $$x_{n-1} + a(x_{n-2} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x1 + ax_0))\ldots))$$

Hash Codes

4. Cyclic-Shift Hash Codes:

A variant of the polynomial hash code

replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits.

5-bit cyclic shift of the 32-bit value

taking the leftmost five bits

00111101100101101010100010101000

placing them on the rightmost side of the representation

10110010110101010001010100000111.

(Python, cyclic shift of bits by careful use of the bitwise operators << and >>, taking care to truncate results to 32-bit integers.)

37

37

Hash Codes

```
def hash_code(s):
  mask = (1 << 32) − 1                          # limit to 32-bit integers
  h = 0
  for character in s:
    h = (h << 5 & mask) | (h >> 27)             # 5-bit cyclic shift of running sum
    h += ord(character)                         # add in value of next character
  return h
```

38

38

Hash Codes in Python

- A built-in function with signature hash(x) that returns an integer value that serves as the hash code for object x.
- Only immutable data types are deemed hashable in Python.
  - Among Python's built-in data types, the immutable int, float, str, tuple, and frozenset classes produce robust hash codes, via the hash function, using techniques similar to those discussed earlier in this section.
  - Hash codes for character strings are well crafted based on a technique similar to polynomial hash codes, except using exclusive-or computations rather than additions.

39

39

Hash Codes in Python

- Hash codes for tuples:
  - Combination of the hash codes of the individual elements of the tuple.
  - Exclusive-or of the individual hash codes without any shifting.
- If hash(x) is called for an instance x of a mutable type, such as a list, a TypeError is raised
- Instances of user-defined classes are treated as unhashable by default, with a TypeError raised by the hash function.
- A function that computes hash codes can be implemented in the form of a special method named hash within a class.

40

40

Hash function for color class

- Color class : red, green, and blue

```
def __hash__ (self):
        return hash( (self. red, self. green, self. blue) )
        # hash combined tuple
```

- Any implementation of hash must be consistent:
  - if x == y, then hash(x) == hash(y).
  - This rule extends to any well-defined comparisons between objects of different classes.
    - For example, Since Python treats the expression 5 == 5.0 as true, it ensures that hash(5) and hash(5.0) are the same.

Compression Functions

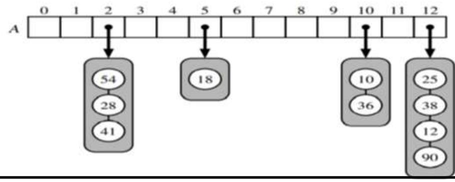
- The Division Method:
  - k mod N,

  where N, the size of the bucket array, is a fixed positive integer.
- Multiply-Add-and-Divide (or "MAD") method.
  - [(ai+b) mod p] mod N

  where N is the size of the bucket array,
  p is a prime number larger than N,
  and a and b are integers chosen at random from the interval [0, p− 1], with a > 0.

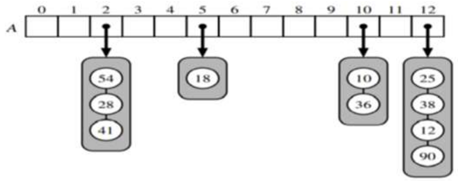
Collision-Handling Schemes

- Separate Chainings:
  - Assuming we use a good hash function to index the n items of our map in a bucket array of capacity N,
    - the expected size of a bucket is n/N.
    - The core map operations run in O(n/N).
    - The ratio λ = n/N, called the load factor of the hash table, should be bounded by a small constant, preferably below 1.
    - As long as λ is O(1), the core operations on the hash table run in O(1) expected time



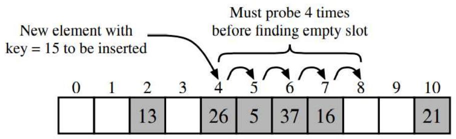
Collision-Handling Schemes

- slight disadvantage:
  - Use of an auxiliary data structure—a list—to hold items with colliding keys. (-) a program for a small handheld device
- Open addressing :
  - requires that the load factor is always at most 1
  - items are stored directly in the cells of the bucket array itself.



Linear Probing and Its Variants

- if we try to insert an item (k,v) into a bucket A[j] that is already occupied, where j = h(k),
  - then we next try A[(j +1) mod N].
  - If A[(j +1) mod N] is also occupied,
    - then we try A[(j + 2) mod N],
    - and so on, until we find an empty bucket that can accept the new item



Insertion into a hash table with integer keys using linear probing. The hash function is $h(k) = k \bmod 11$. Values associated with keys are not shown.

Linear Probing and Its Variants

- Quadratic probing :
  - Iteratively tries the buckets A[(h(k)+ f(i)) mod N], for i = 0,1,2,..., where f(i) = i^2, until finding an empty bucket.
    - creates secondary clustering, where the set of filled array cells still has a non-uniform pattern
    - When N is prime and the bucket array is less than half full, the quadratic probing strategy is guaranteed to find an empty slot.
    - However, this guarantee is not valid once the table becomes at least half full, or if N is not chosen as a prime number;

Linear Probing and Its Variants

- Double hashing strategy:
  - choose a secondary hash function, h'
  - if h maps some key k to a bucket A[h(k)] that is already occupied,
    - Then iteratively try the buckets A[(h(k) + f(i)) mod N] next, for i = 1,2,3,..., where f(i) = i · h'(k).
    - the secondary hash function is not allowed to evaluate to zero; a common choice is :
      - H'(k) = q−(k mod q), for some prime number q < N.
      - N should be a prime.
- Python's dictionary class:
  - Avoid clustering with open addressing
  - Iteratively try buckets A[(h(k) + f(i)) mod N]
  - f(i) is based on a pseudo-random number generator, sequence of subsequent probes that depends upon bits of the original hash code.

Open adressing implementation

```
class ProbeHashMap(HashMapBase):
  """Hash map implemented with linear probing for collision resolution."""
  _AVAIL = object( )        # sentinal marks locations of previous deletions

  def _is_available(self, j):
    """Return True if index j is available in table."""
    return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL

```

Open adressing implementation

```
def _find_slot(self, j, k):
  """Search for key k in bucket at index j.

  Return (success, index) tuple, described as follows:
  If match was found, success is True and index denotes its location.
  If no match found, success is False and index denotes first available slot.
  """
  firstAvail = None
  while True:
    if self._is_available(j):
      if firstAvail is None:
        firstAvail = j                          # mark this as first avail
      if self._table[j] is None:
        return (False, firstAvail)              # search has failed
    elif k == self._table[j]._key:
      return (True, j)                          # found a match
    j = (j + 1) % len(self._table)              # keep looking (cyclically)
```


Open adressing implementation

```
def _bucket_getitem(self, j, k):
  found, s = self._find_slot(j, k)
  if not found:
    raise KeyError('Key Error: ' + repr(k))     # no match found
  return self._table[s]._value

def _bucket_setitem(self, j, k, v):
  found, s = self._find_slot(j, k)
  if not found:
    self._table[s] = self._Item(k,v)            # insert new item
    self._n += 1                                # size has increased
  else:
    self._table[s]._value = v                   # overwrite existing
```


Open adressing implementation

```
def _bucket_delitem(self, j, k):
  found, s = self._find_slot(j, k)
  if not found:
    raise KeyError('Key Error: ' + repr(k))     # no match found
  self._table[s] = ProbeHashMap._AVAIL          # mark as vacated

def __iter__(self):
  for j in range(len(self._table)):             # scan entire table
    if not self._is_available(j):
      yield self._table[j]._key
```


Separate Chainings implementation

```
class ChainHashMap(HashMapBase):
  """Hash map implemented with separate chaining for collision resolution."""

  def _bucket_getitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))   # no match found
    return bucket[k]                            # may raise KeyError

  def _bucket_setitem(self, j, k, v):
    if self._table[j] is None:
      self._table[j] = UnsortedTableMap( )      # bucket is new to the table
    oldsize = len(self._table[j])
    self._table[j][k] = v
    if len(self._table[j]) > oldsize:           # key was new to the table
      self._n += 1                              # increase overall map size
```

Separate Chainings implementation

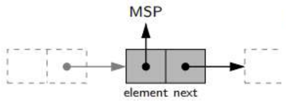
```python
def _bucket_delitem(self, j, k):
  bucket = self._table[j]
  if bucket is None:
    raise KeyError('Key Error: ' + repr(k))      # no match found
  del bucket[k]                                  # may raise KeyError

def __iter__(self):
  for bucket in self._table:
    if bucket is not None:                       # a nonempty slot
      for key in bucket:
        yield key
```

53

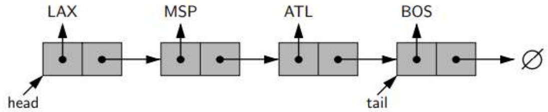
53

Separate Chainings implementation

Implement separate chainings par single linked list




```python
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'          # streamline memory usage

  def __init__(self, element, next):      # initialize node's fields
    self._element = element               # reference to user's element
    self._next = next                     # reference to next node
```



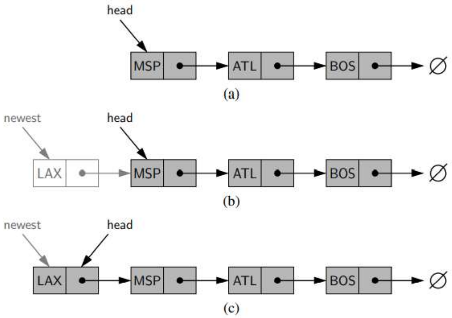
Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that identifies the first node of the list, and in some applications another member named tail that identifies the last node of the list. The **None** object is denoted as Ø.

54

54

Separate Chainings implementation

- Implement separate chainings par single linked list



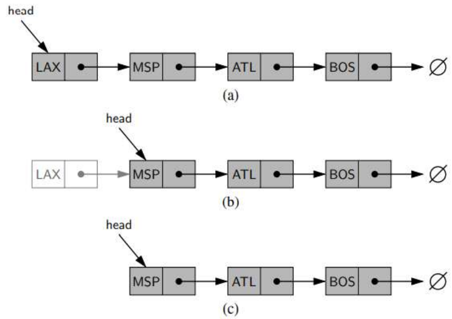
Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the head reference.

55

55

Separate Chainings implementation

- Implement separate chainings with single linked list



Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

56

56

Separate Chainings implementation

- Implement separate chainings with single linked list

```
class Linked
  #------------------------- nested _Node class -------------------------
  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    __slots__ = '_element', '_next'         # streamline memory usage

    def __init__(self, element, next):      # initialize node's fields
      self._element = element               # reference to user's element
      self._next = next                     # reference to next node

  def __init__(self):

    self._head = None                       # reference to the head node
    self._size = 0                          # number of       elements

  def __len__(self):
    """Return the number of elements        """
    return self._size
```

57

57

Separate Chainings implementation

- Implement separate chainings with single linked list

```
def push(self, e):
  """Add element e                 """
  self._head = self._Node(e, self._head)    # create and link a new node
  self._size += 1


def _delete_node(self, e, predecessor           ):
  predecessor._next = node._next                          !
  self._size -= 1
  element = node._element
  node._element = None       # deprecate node
  return element
                                                        #
```

58

58

Exercises

1. Give a concrete implementation of the pop method in the context of the MutableMapping class, relying only on the five primary abstract methods of that class.
2. Give a concrete implementation of the items( ) method in the context of the MutableMapping class, relying only on the five primary abstract methods of that class.
3. Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?

59

59

Exercises

3. What would be a good hash code for a vehicle identification number that is a string of numbers and letters of the form "9X9XX99X9XX999999," where a "9" represents a digit and an "X" represents a letter?
4. Computing a hash code can be expensive, especially for lengthy keys. In our hash table implementations, we compute the hash code when first inserting an item, and recompute each item's hash code each time we resize our table. Python's dict class makes an interesting trade-off. The hash code is computed once, when an item is inserted, and the hash code is stored as an extra field of the item composite, so that it need not be recomputed. Reimplement our HashTableBase class to use such an approach

60

60

Problem

Key word extraction in text documents: implement Text Rank and its variants

- Prepare data
- Word frequency in documents (dict)
- Construct Graph of word co-occurrency (sparce matrix)
- Apply Page Rank to retrive key word