Lecture 11: Dictionaries

Stephen Huang

April 13, 2023

Note

- Due to the time limitation, we will skip the chapter on Tuples.
- However, we will briefly compare lists and tuples before discussing dictionaries.
- Lists and tuples are the commonly used data structures in Python.
- But what are the similarities and differences between them?

Contents

1. Lists and Tuples

- Both lists and tuples consist of objects which can be referenced by their position number within the object.
- We can change the elements, add extra elements, or delete an element with a list.
- You're probably wondering why the two structures are provided - are they both necessary?
- A tuple is much more efficient in use after it has been created.

Tuples

- Lists and tuples are containers where data can be accessed easily.
- Lists and tuples are also sequences in that data are organized in a well-defined sequential manner.
- Tuples are immutable, whereas lists are mutable. We pay the price for the efficiency of the mutability of lists.
- Both are indexed by integers starting with 0.

Tuples

- A tuple is a comma-separated list of values.
- Although it is unnecessary, enclosing tuples in parentheses is common.
- To create a tuple with a single element, you must include a final comma.
- To create a tuple, we can also use the built-in function tuple().
  - If the argument is a sequence (string, list, or tuple), the result is a tuple with the sequence elements.

Summary

	List	Tuple
Syntax	Square brackets **[]**	Parentheses **()**
Mutability	Mutable	**Im**mutable
Built-in Methods	**More** methods such as pop(), insert(), append()	**Fewer** methods
Storage Efficiency	Consumes **more** memory/storage	Consumes **less** memory/storage
Time Efficiency	**Slower** to create list and to access list elements	**Faster** to create tuple and to access tuple elements

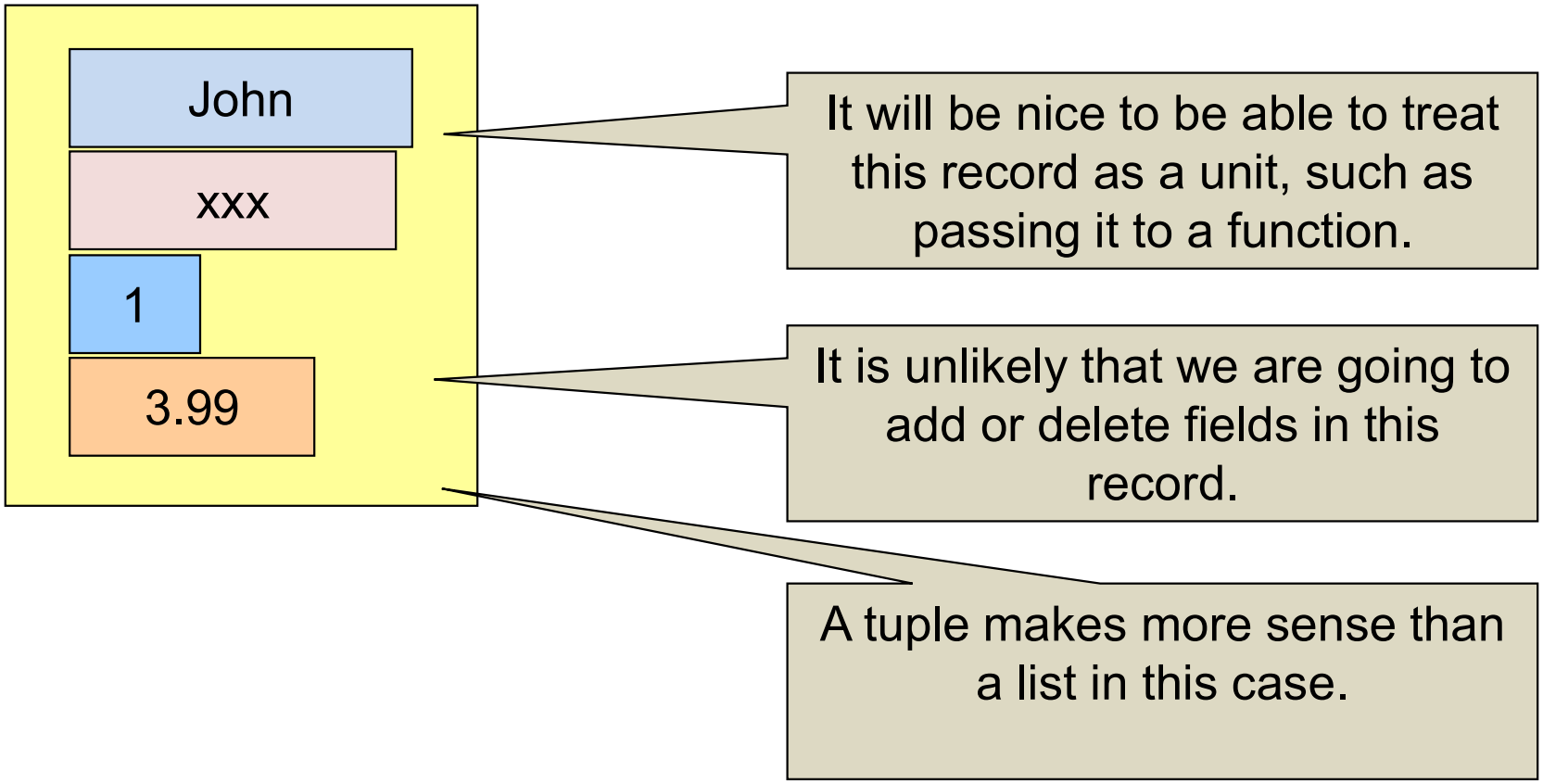
Example: lists

Tire together by the index

	Name	telNum	classCode	GPA
0	John	xxx	1	3.99
1	Bob	xxx	2	2.01
2	Amy	xxx	3	3.50
3	Joe	xxx	4	3.22
4	June	xxx	1	2.91
5	May	xxx	2	2.50

This is a logical unit that we would like to keep together.

"Record"


John
xxx
1
3.99
It will be nice to be able to treat this record as a unit, such as passing it to a function.
It is unlikely that we are going to add or delete fields in this record.
A tuple makes more sense than a list in this case.

2. Dictionary Basics

- Dictionaries share some syntactic properties with lists and tuples, but significant differences exist.
- All the keys/words are alphabetically arranged in a traditional printed dictionary. There is no such ordering of the keys in Python's dict.

Dictionary (Dict)

- A dictionary is a collection that is
  - unordered,
  - changeable, and
  - indexed.
- In Python, dictionaries are written with curly brackets {}, with keys and values.

Dictionary Basics

- Python provides a container known as a dictionary, or dict for short.
- Dictionaries share some syntactic properties with lists and tuples, but significant differences exist.
- Dictionaries are not sequential collections of data. Instead, dictionaries consist of key-value pairs.
- To obtain a "value" (i.e., the data of interest), you specify its associated key.

Dictionary

- The order in which Python stores the key-value pairs is not a concern.
- We need to know that when we specify the key ’name’, Python will provide the associated value. Key -> value.
- In a traditional dictionary model, we provide a word (as the key), and the dictionary returns the word’s definition (as the value).
- You can also model a dictionary as a list of tuples, but you have to do your search.

A Comparison

Index → Value

Key → Value

0	Value0
1	Value1
2	Value2
3	Value3
4	Value4
5	Value5

A Comparison

Index → Value

Key → Value

(Key, Value)

Key0	Value0
Key1	Value1
Key2	Value2
Key3	Value3
Key4	Value4

Examples

‘Name’	‘Python’
‘Credit’	3
‘Room’	232
‘Building’	‘PGH’

‘brand’	‘Ford’
‘model’	‘Mustang’
‘year’	1964

Basics

- To create a dictionary, we use curly braces {}.
- Start with an empty dictionary; we can add a key-value pair to a dictionary.
- A list of key-value pairs can be specified to initialize a dictionary.
- A comma (,) is used to separate key-value pairs.
- A colon (:) is used to separate key and value.
- A pair of brackets ([key]) is used to access the value of a given key.

Adding A Key-Value Pair

- How to add a new pair to a dictionary?
- The following statement adds a new pair to the dictionary d.

```
d[key] = value
```

- If there is a key-value pair with the same key in the dict, the value will be over-written.
  - You cannot have two key-value pairs with the same key.
- If there is no pair with the key, one will be created.

Example

```
myclass ={'Smith':    99,
          'Johnson':  88,
          'Johnsson': 77,
          'Huang':    88}

for stu in myclass:
    print(f'{stu} {myclass[stu]}')
print()
```

Example

```
phone_book = {'Smith':    '713-743-3350',
              'Johnson':  '713-743-3334',
              'Johnsson': '713-743-3388',
              'Huang':    '713-743-3338'}

for fac in phone_book:
    print(f'{fac}: {phone_book[fac]}')
```

Example

```
myclass = {'name':     'Python',
           'credit':   3,
           'room':     232,
           'building': 'PGH'}
```

```
print(myclass)
```

```
{'name': 'Python', 'credit': 3, 'room': 232, 'building': 'PGH'}
```

```
print(type(myclass))
```

```
<class 'dict'>
```

```
print(myclass['name'])
```

```
'Python'
```

Examples

```
print(myclass['name'])
'Python'

print(myclass['Name'])
  Traceback (most recent call last):
    File "<pyshell#16>", line 1, in <module>
      myclass['Name']
  KeyError: 'Name'
```

Dict

- Key is case-sensitive.
- Because of the {}, the statement can span several lines.

```
>>>
>>> fall2017 = {'name': 'Algorithm',
                'credit': 3,
                'room': 200,
                'building': 'PGH'}
>>> fall2017
{'name': 'Algorithm', 'credit': 3, 'room': 200,
'building': 'PGH'}
>>>
```

Initialization

```
d = {
    <key>: <value>,
    <key>: <value>,
    .
    .
    .
    <key>: <value>
}
```

Keys

- As shown in our examples, keys do not have to be of string type.
- Keys can be numeric values, tuples, or any immutable object.
- So, you can use an integer as an index, and it does not have to start with 0, and they don't have to be consecutive numbers.
- There is no guarantee that the pairs will be stored in the order of the keys.
  - The older version may sort it.
  - The newer version of Python remembers the order of entry.

A Terrible Example

```
# No one in the right mind will do this.

d = {
    'alma mater': 'UH',
    42: 'The meaning of Life.',
    (3,4): {'first': 33, 'second': 3+4},
    5.71: [5, 0.71]
}
```

Example

```
huang = {'name':'Huang', 'phone':'x3-3338'}
cs1336 = {'name':'Python', 'courseNum':1336,
'instructor': huang}

print(cs1336)
print("\nKeys: ", cs1336.keys())
print("\nValues: ", cs1336.values())
print("\nItems: ", cs1336.items())
print("\nGet courseNum: ",
                    cs1336.get('courseNum'))
```

Example

```
{'name':'Python', 'courseNum':1336, 'instructor':
{'name':'Huang', 'phone':'x3-3338'}}

Keys: dict_keys(['name', 'courseNum', 'instructor'])

Values: dict_values(['Python', 1336, {'name':
'Huang', 'phone': 'x3-3338'}])

Items: dict_items([('name', 'Python'), ('courseNum',
1336), ('instructor', {'name': 'Huang', 'phone':
'x3-3338'})])

Get courseNum: 1336
```

3. Cycling through Dictionary

- Dict is an "iterable" and can be used in a for-loop header.
- Iterator provides an easy way to go through the dictionary and "process" each item.
- The iterator is equal to the key of the dict.
- You can use the iterator to access the values using `dict[key]`.

Example

```
cs1336 = {
    'name':'Python',
    'credit':3,
    'room':232,
    'building':'PGH'}

for key in cs1336:
    print(key)
print()
```

```
name
credit
room
building
```

Example

```
president = {
   41: 'George H. W. Bush',
   42: 'Bill Clinton',
   43: 'George W. Bush',
   44: 'Barack Obama',
   45: 'Donald Trump',
   36: 'Lyndon B. Johnson'}
```

```
{41: 'George H. W. Bush', 42: 'Bill
Clinton', 43: 'George W. Bush', 44:
'Barack Obama', 45: 'Donald Trump',
36: 'Lyndon B. Johnson'}
```

Example

```
for prez in president:
    print(prez)
print()
```

```
41
42
43
44
45
36
```

Example

```
for prez in president:
    print(f"{prez}: {president[prez]}")
print()
```

```
41:  George H. W. Bush
42:  Bill Clinton
43:  George W. Bush
44:  Barack Obama
45:  Donald Trump
36:  Lyndon B. Johnson
```

Example

```
for prez in sorted(president):
    print(prez, president[prez], sep = '')
```

```
36:  Lyndon B. Johnson
41:  George H. W. Bush
42:  Bill Clinton
43:  George W. Bush
44:  Barack Obama
45:  Donald Trump
```

Example

```
for value in president.values():
    print(value)
```

```
George H. W. Bush
Bill Clinton
George W. Bush
Barack Obama
Donald Trump
Lyndon B. Johnson
```

Get(key) vs. [key]

- The `get()` method can look up values for a given key.
- But so is dict[key].
- What's the difference?
  - If the key does not exist, the get() method returns None.
  - An error will result if we try to access the value using dict[key] for a non-existing key.

Get()

```
cs1336 = {'name': 'Python',
          'courseNum': 1336,
          'instructor': 'Huang'}

cs1430 = {'name': 'C++',
          'courseNum': 1430} # no instructor

for key in ['name', 'courseNum', 'instructor']:
    print(cs1336[key])

for key in ['name', 'courseNum', 'instructor']:
    print(cs1336.get(key))
```

Default Return Value

```
for key in ['name', 'courseNum', 'instructor']:
    print(cs1430.get(key, 'John Doe'))

for key in ['name', 'courseNum', 'instructor']:
    print(cs1430[key])
```

```
C++
1430
John Doe
```

(None w/o default value)

```
C++
1430
KeyError: 'instructor'
```

Dict as a Dictionary

- Most examples up to this point are using dict to store a "record", such as a car, a student, or a class.
- Using a dict is slightly better than a list because you can use a 'model' instead of a meaningless index.
- In the car example, 'brand', 'model', and 'year' are three different types of values.

Dictionary with a Key

- In a real dictionary, we use a word to search for the meaning of the word.
- Suppose we want to use a car model to search for the brand of the car. We can easily store many such "words".

```
{'Mustang': 'Ford', 'Edge': 'Ford',
 'Camry': 'Toyota', 'Bronco': 'Ford'}
```

Example

```
car_cat = {
    'Dodge Charger': 29995,
    'Toyota Camry':  24425,
    'Honda Civic':   19850,
    'BMW E':         54000,
    'Kia Forte':     17890,
    'Mercedes S':    94250,
    'Ford Fusion':   23170
}
```

Example

```
stu = {
      101: "Huang, Stephen",
      102: "Johnson, Olin",
      190: "Smith, John",
      123: "Anderson, Robert"
}
```

Example

```
stu = {
    101: {'name':"Huang, Stephen",    'major':"CS"},
    102: {'name':"Johnson, Olin",     'major':"CS"},
    190: {'name':"Smith, John",       'major':"Bio"},
    123: {'name':"Anderson, Robert", 'major':"Math"}
}
```

Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

4. Dictionary as a Counter

- This section discusses using dict as a counter for certain keys.
- Suppose you are given a string, and you want to count how many times each letter or word appears.

Example

```
def histogram(s):
    d = dict()
    for ch in s:
        if ch not in d:
            d[ch] = 1
        else:
            d[ch] += 1
    return d

h = histogram('mississippi')
print(h)
```

```
{'m': 1, 'i': 4, 's': 4, 'p': 2}
```

Counter

- The get() method can obtain the value associated with a key.
- If the key does not exist, get() returns None by default.
- However, an optional argument can be provided to specify the return value for a non-existent key.
  - If it is not in there, add it in.

Get + default

```
def histogram(s):
    d = dict()
    for ch in s:
        d[ch] = d.get(ch, 0) + 1
    return d
```

Comparison

```
def histogram(s):
    d = dict()
    for ch in s:
        if ch not in d:
            d[ch] = 1
        else:
            d[ch] += 1
    return d

def histogram(s):
    d = dict()
    for ch in s:
        d[ch] = d.get(ch, 0) + 1
    return d
```

Reverse Lookup

- The dictionary is not designed for searching for an item with a particular value.
- There may be multiple items with the same value.
- It takes some effort to do that.

Example

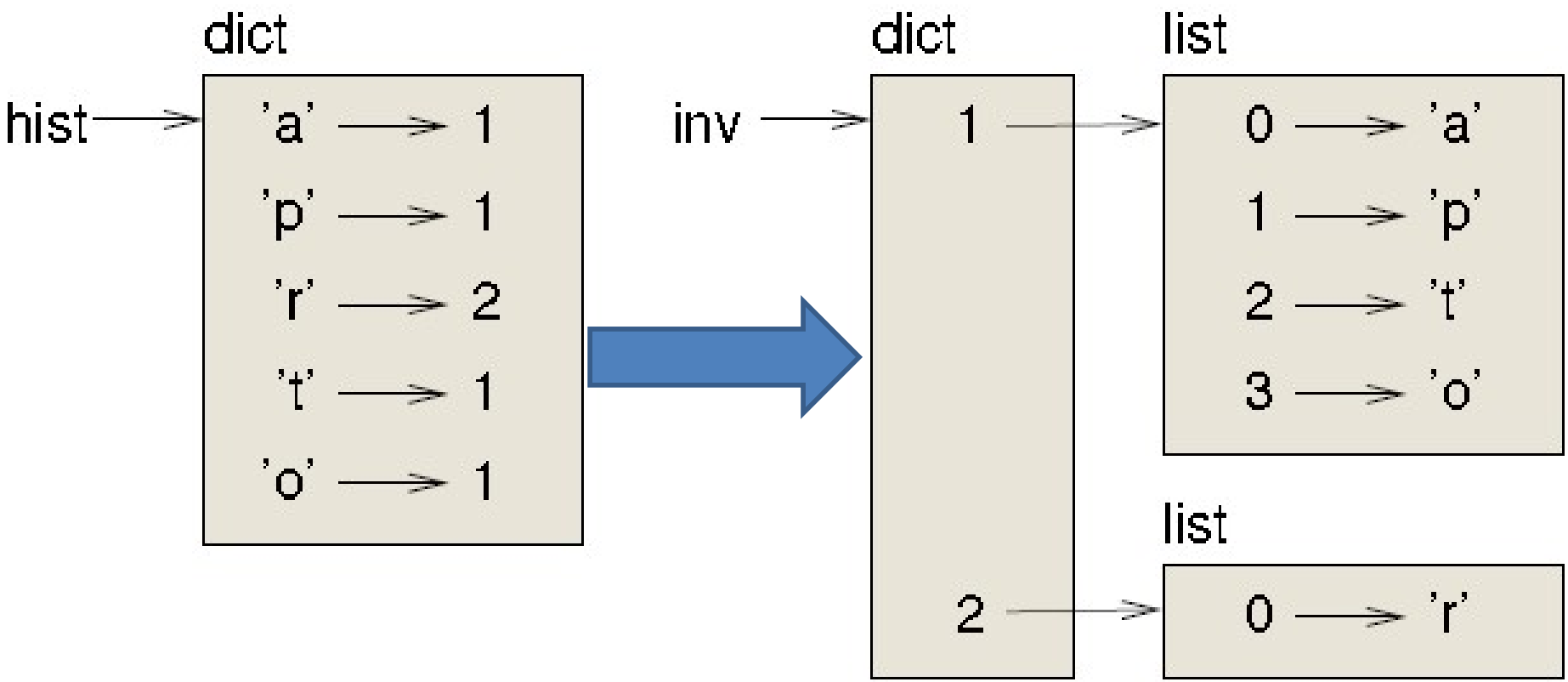
```
def rev_get(d, v):
    for key in d:
        if d.get(key) == v:
            return key

h = histogram('mississippi')

print(rev_get(h, 2))
```

```
{'m': 1, 'i': 4, 's': 4, 'p': 2}
p
```

5. Dictionary of Lists

- Dictionary values can be of any type, including list and dictionary itself.
- We have seen an example of a value of dictionary type (instructor).
- Let's look at one example with a list as one value type.
- Let's create a dictionary that maps from frequencies to letters, with each value in the inverted dictionary a list of letters.

State Diagram



parrot

Invert

```
def invert_dict(d):
    inverse = dict()
    for key  in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

Test

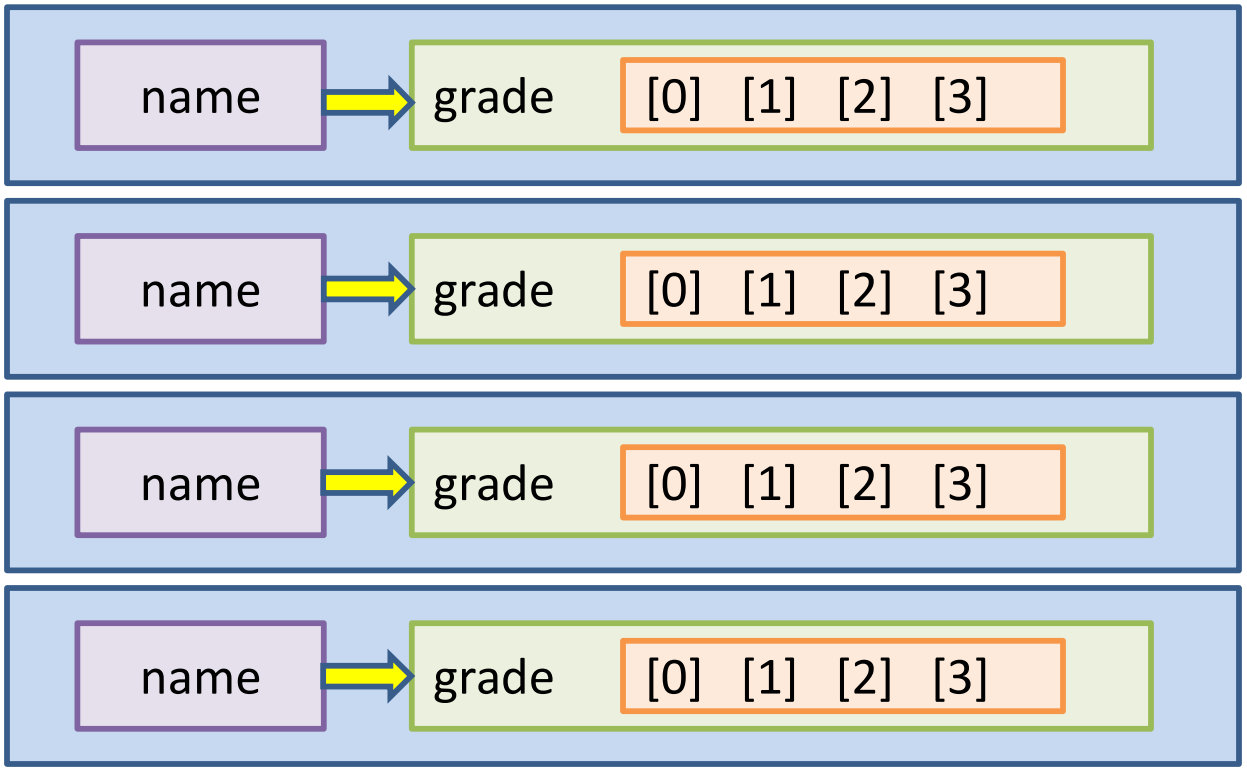
```
def print_dict(d):
    print("Dict:")
    for key in sorted(d):
        print("  ", key, "->", d[key])
    print()

h = histogram('parrot')
print_dict(h)
d = invert_dict(h)
print_dict(d)
```

```
Dict:
   a -> 1
   o -> 1
   p -> 1
   r -> 2
   t -> 1

Dict:
   1 -> ['p','a','o','t']
   2 -> ['r']
```

6. List of Dictionaries

- Given a list of student records in a text file, one record per line, store the student record as a dict with a name (string) and four grades (as a list of numbers).
- The grades list is part of the dictionary.
- Build a list of the student records, i. e., a list of dictionaries.

Structure


name
grade
[0] [1] [2] [3]
name
grade
[0] [1] [2] [3]
name
grade
[0] [1] [2] [3]
name
grade
[0] [1] [2] [3]

function

```
def build_roster(f):
    roster = []
    i = 0
    for line in f:
        d = dict()
        list = line.split()
        d['name'] = list.pop(0)
        d['grade'] = list_a2i(list)
        roster.append(d)
        i += 1
    return roster
```

functions

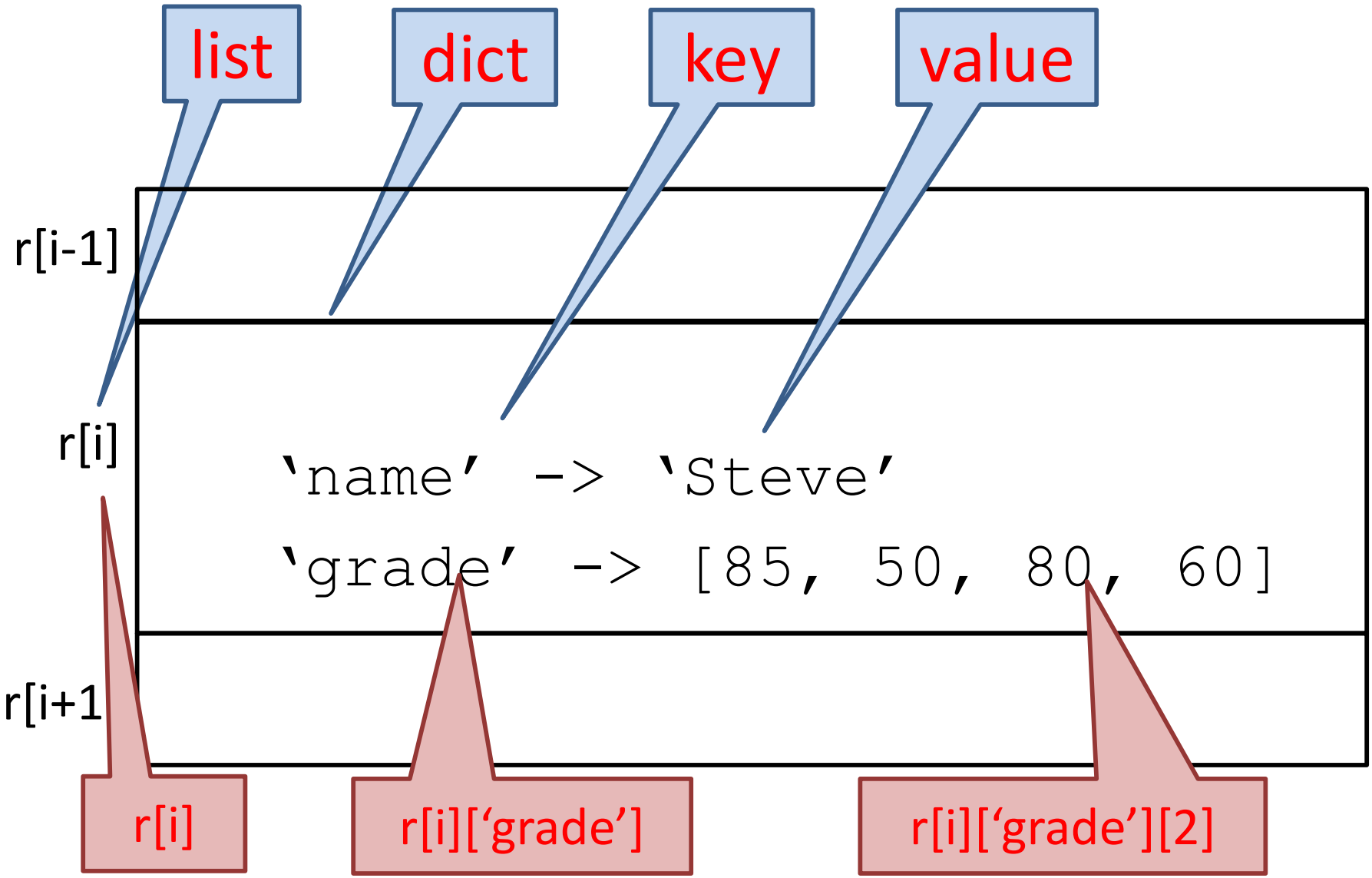
```
def list_a2i(list):
    for i in range(len(list)):
        list[i] = int(list[i])
    return list

def print_dict(d):
    print("Dict:")
    for key in d:
        print("  ", key, "->", d[key])
    print()
```

main

```
f = open("grades.txt")
r = build_roster(f)
for d in r:
    print_dict(d)
```

```
Dict:
   name -> Steve
   grade -> [85, 50, 80, 60]

Dict:
   name -> James
   grade -> [100, 100, 95, 90]

Dict:
   name -> Zack
   grade -> [99, 95, 95, 100]

Dict:
   name -> Jackie
   grade -> [95, 45, 90, 80]

Dict:
   name -> Liz
   grade -> [80, 70, 90, 70]
```

One level at a time



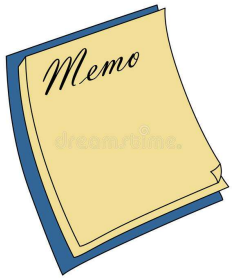
Find the lowest value

```
low = 100
for d in r:
    for i in range(4):
        if d['grade'][i]<low:
            low = d['grade'][i]
```

```
r
r[j] = d
r[j]['grade']
r[j]['grade'][i]
```

7. Memo

- For efficiency reasons, we should avoid computing values that have been computed before.
- One solution is to keep track of values that have already been computed by storing them in a dictionary.
- A previously computed value stored for later use is called a memo.
- The recursive Fibonacci function is a simple example of using a memo to save computation time.



Code

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Code

```
def fib(n):
    if n in memo:
        return memo[n]

    result = fib(n-1)+fib(n-2)

    memo[n] = result

    return result
```

If saved in memo

Reuse the value

If not, compute the value

Save the value in memo

memo ={0:0, 1:1}

Overview

Types	Ordered	Indexed	Collection Changeable?	Item Changeable?	Duplicate
List	Yes	Yes	Add/Remove	Yes	Yes
Tuple	Yes	Yes	No	No	Yes
Set	No	No	Add/Remove	No	No
Dictionary	No	Yes	Add/Remove	Value Yes Key No	No