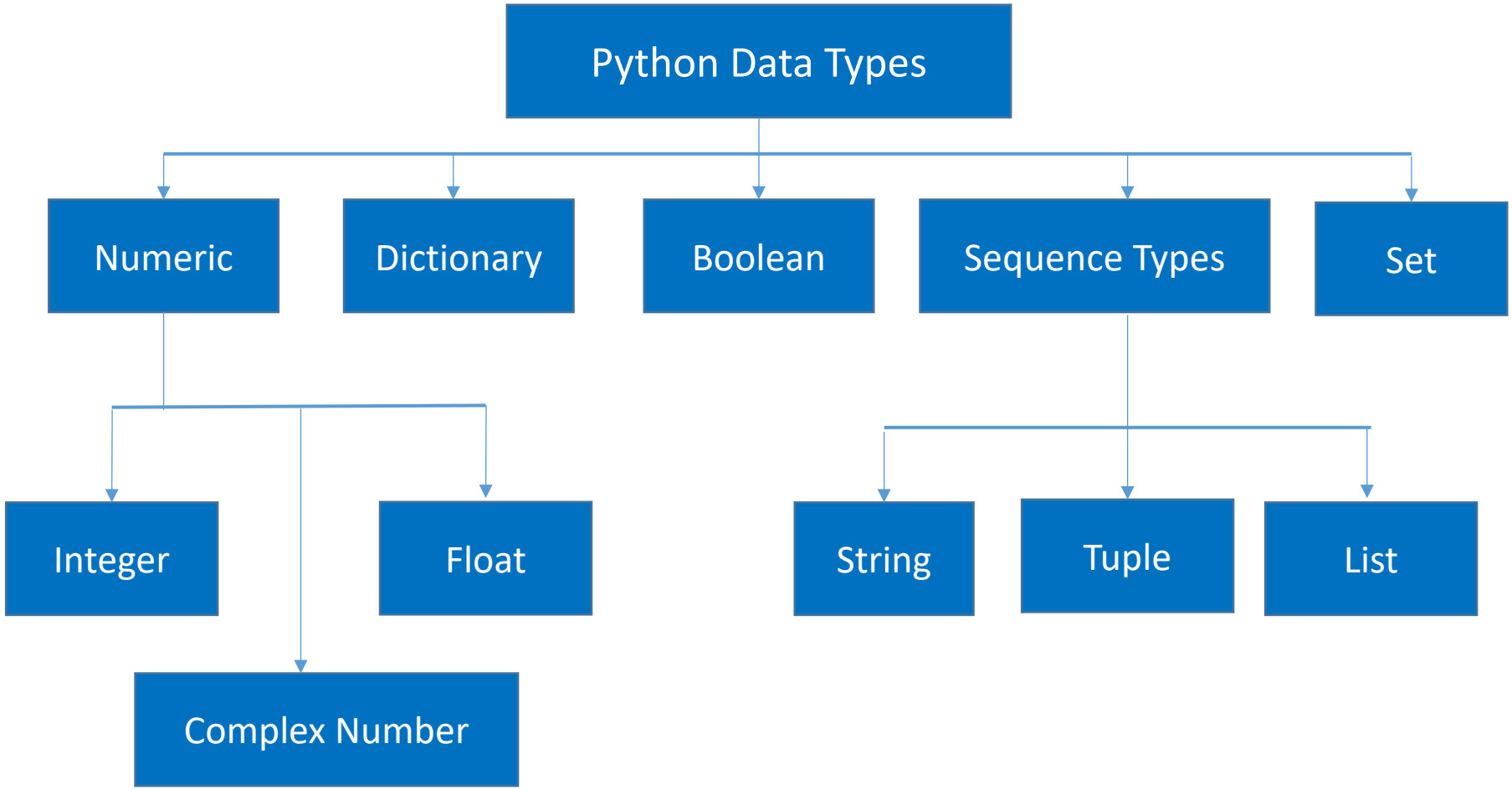
LẬP TRÌNH VỚI PYTHON

(Programming with Python)

Th.S Nguyễn Hoàng Thành

Email: thanhnh@ptithcm.edu.vn

Tel: 0909 682 711


Python Data Types
Numeric
Dictionary
Boolean
Sequence Types
Set
Integer
Float
Complex Number
String
Tuple
List

10. Python - Lists

- Lists are Python's most flexible ordered collection object type.
- Lists can contain any sort of object:
  - numbers,
  - strings,
  - and even other lists.
- Lists may be changed in place by assignment to offsets and slices, list method calls, deletion state- ments, and more—they are ***mutable*** objects.

Python lists are:

- *Ordered collections of arbitrary objects*
- *Accessed by offset*
- *Variable-length, heterogeneous, and arbitrarily nestable*
- *Of the category "mutable sequence"*
- *Arrays of object references*

Common list literals and operations (Table 8.1, p.248 – 249)

Operation	Interpretation
L = []	An empty list
L = [123, 'abc', 1.23, {}]	Four items: indexes 0..3
L = ['Bob', 40.0, ['dev', 'mgr']]	Nested sublists
L = list('spam')	List of an iterable's items, list of successive integers
L = list(range(-4, 4))	
L[i]	Index, index of index, slice, length
L[i][j]	
L[i:j]	
len(L)	

Common list literals and operations (Table 8.1, p.248 – 249)

Operation	Interpretation
L1 + L2	Concatenate, repeat
L*3	
for x in L: print(x)	Iteration, membership
3 in L	
L.append(4)	Methods: growing
L.extend([5,6,7])	
L.insert(i, X)	
L.index(X)	Methods: searching
L.count(X)	

Common list literals and operations (Table 8.1, p.248 – 249)

Operation	Interpretation
L.sort()	Methods: sorting, reversing, copying (3.3+), clearing (3.3+)
L.reverse()	
L.copy()	
L.clear()	
L.pop(i)	Methods, statements: shrinking
L.remove(X)	
del L[i]	
del L[i:j]	
L[i:j] = []	

Common list literals and operations (Table 8.1, p.248 – 249)

Operation	Interpretation
L[i] = 3	Index assignment, slice assignment
L[i:j] = [4,5,6]	
L = [x**2 for x in range(5)]	List comprehensions and maps
list(map(ord, 'spam'))	

Basic List Operations

Lists respond to the + and * operators much like strings

```
>>> len([1, 2, 3, 4])                    # Length
4
>>> [1, 2, 3] + [4, 5, 6]                #Concatenation
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                          #Repetition
['Python!', 'Python!', 'Python!', 'Python!']
>>> [1, 2, 3] + ['Python', '!']
[1, 2, 3, 'Python', '!']
```

List Iteration and Comprehensions

- Lists respond to all the sequence operations we used on strings

```
>>> 3 in [1, 2, 3]                          # Membership
True
>>> for x in [1, 2, 3]:
...         print(x, end=' ')                # Iteration (2.X uses: print x,)
...
1 2 3
```

- List comprehensions are a way to build a new list by applying an expression to each item in a sequence (really, in any iterable), and are close relatives to for loops

```
>>> res = [c * 4 for c in 'SPAM']                    # List comprehensions
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

```
>>> res = []
>>> for c in 'SPAM':                    # List comprehension equivalent
...     res.append(c * 4)
...
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

Indexing, Slicing, and Matrixes

- Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                                  # Offsets start at zero
'SPAM!'
>>> L[-2]                                 # Negative: count from the right
'Spam'
>>> L = [1:]
['Spam', 'SPAM!']
```

- Because you can **nest lists** and **other object types** within lists, you will sometimes need to string together index operations to go deeper into a data structure.

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
```

Changing Lists in Place

- Because lists are mutable, they support operations that change a list object *in place*.
- Python deals only in object references, this distinction between changing an object in place and creating a new object matters;

Index and slice assignments

- When using a list, you can change its contents by assigning to either a particular item (offset) or an entire section (slice):

```
>>> L = ['eggs,', 'milk', 'beer', 'diapers']
>>> L[1] = 'paper'                          # Index assignment
>>> L
['eggs,', 'paper', 'beer', 'diapers']
>>> L[0:2] = ['eat', 'more']                # Slice assignment: delete+insert
>>> L                                       # Replaces items 0,1
['eat', 'more', 'beer', 'diapers']
```

Slice assignment

```
>>> L[0:2] = ['eat', 'more']       # Slice assignment: delete+insert
>>> L                              # Replaces items 0,1
```

- The last operation in the preceding example, replaces an entire section of a list in a single step. Because it can be a bit complex, it is perhaps best thought of as a combination of two steps:
  - *Deletion*. The slice you specify to the left of the = is deleted.
  - *Insertion*. The new items contained in the iterable object to the right of the = are inserted into the list on the left, at the place where the old slice was deleted.

- Insert First
- Insert Last
- Insert at postion

```
>>> L = [2, 3, 4]
>>> L[:0] = [0, 1]          # Insert all at :0, an empty slice at front
>>> L
[0, 1, 2, 3, 4]
>>> L[len(L):] = [5, 6, 7]  # Insert all at len(L):, an empty slice at end
>>> L
[0, 1, 2, 3, 4, 5, 6, 7]
>>> L.extend([8, 9, 10])    # Insert all at end, named method
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Python list objects also support type-specific method calls, many of which change the subject list in place:

```
>>> L = [2, 3, 4]
>>> L.append(0)                    # Append method call: add item at end
>>> L
[2, 3, 4, 0]
>>> L.sort()                       # Sort list items
>>> L
[0, 2, 3, 4]
```

More on sorting lists

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                                  # Sort with mixed case
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                     # Normalize to lowercase
>>> L
['abc', 'ABD', 'aBe']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)       # Change sort order
>>> L
['aBe', 'ABD', 'abc']
```

Other common list methods

```
>>> L = [1, 2, 3, 4, 5]
>>> L.pop()                    # Delete and return last item (by default: −1)
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                # In-place reversal method
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))          # Reversal built-in with a result (iterator)
[1, 2, 3, 4]
```

```
>>> L = ['spam', 'eggs', 'ham']
>>> L.index('eggs')                    # Index of an object (search/find)
1
>>> L.insert(1, 'toast')               # Insert at position
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove('eggs')                   # Delete by value
>>> L
['spam', 'toast', 'ham']
>>> L.pop(1)                           # Delete by position
'toast'
>>> L.count('spam')                    # Number of occurrences
1
```

Other common list operations

- Because lists are mutable, you can use the del statement to delete an item or section in place:

```
>>> L = ['spam', 'eggs', 'ham', 'toast']
>>> del L[0]                                    # Delete one item
>>> L
['eggs', 'ham', 'toast']

>>> del L[1:]
>>> L                                           # Delete an entire section

['eggs']
```

Summary

- List is **created** by placing elements inside square brackets **[]**, separated by commas.
- Access List Elements
  - List Index
  - Negative indexing
- List Slicing in Python
- Add/Change List Elements
- Delete List Elements
- Python List Methods: append(), extend(), insert(), remove(), …

Summary

- List is **created** by placing elements inside square brackets **[]**, separated by commas.

```
>>> myList = [1, 2, 3]                    # list of integers
>>> myList = []                           # empty list
>>> my_list = [1, "Hello", 3.4]           # list with mixed data types
>>> my_list = ["mouse", [8, 4, 6], ['a']] # nested list
```

Access List Elements

- **List Index:**
  - We can use the index operator **[]** to access an item in a list.
  - In Python, indices start at 0.
  - Trying to access indexes other than these will raise an IndexError.
  - The index must be an integer. We can't use float or other types, this will result in **TypeError**
  - Nested lists are accessed using nested indexing.
- **Negative indexing**
  - Python allows negative indexing for its sequences.
  - The index of -1 refers to the last item, -2 to the second last item and so on.

List Slicing in Python

- We can access a range of items in a list by using the slicing operator **:**

```
>>> myList = ['P','T','I', 'T',' ','H','C','M']      # List slicing in Python

>>> print(myList[2:5])                               # elements from index 2 to index 4

>>> print(myList[5:])                                # elements from index 5 to end

>>> print(myList[:])                                 # elements beginning to end
```

Add/Change List Elements

- We can use the assignment **operator =** to change an item or a range of items.
- We can add **one item** to a list using the **append()** method or
- add **several items** using the **extend()** method.
- We can also use **+ operator** to **combine** two lists. This is also called concatenation.
- The *** operator** repeats a list for the given **number of times**.

Delete List Elements

- We can delete one or more items from a list using the Python ***del* statement**. It can even delete the list entirely.
- We can use **remove()** to remove the given item or pop() to remove an item at the given index.
- The **pop()** method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).
- And, if we have to empty the whole list, we can use the **clear()** method.
- Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

Exercise 1.0

- Write a Python code in terminal or script file to sum all the items in a list.

```
>>> List1 = [1, 2, 3, 4, 5, 6]
>>> sum = 0
>>> for x in List1:
...     sum+=x
...
>>> print(sum)
21
```

Exercise 1.1

- Write Python code in terminal or script file to sum all if element is numeric

```
>>> List2 = ["1", 2, "3", 4, 5, 6]
>>> sum = 0
>>> for x in List1:
...         if isinstance(x, (int, float)):
...                 sum+=x
…
>>> print(sum)
17
```

Exercise 2.0

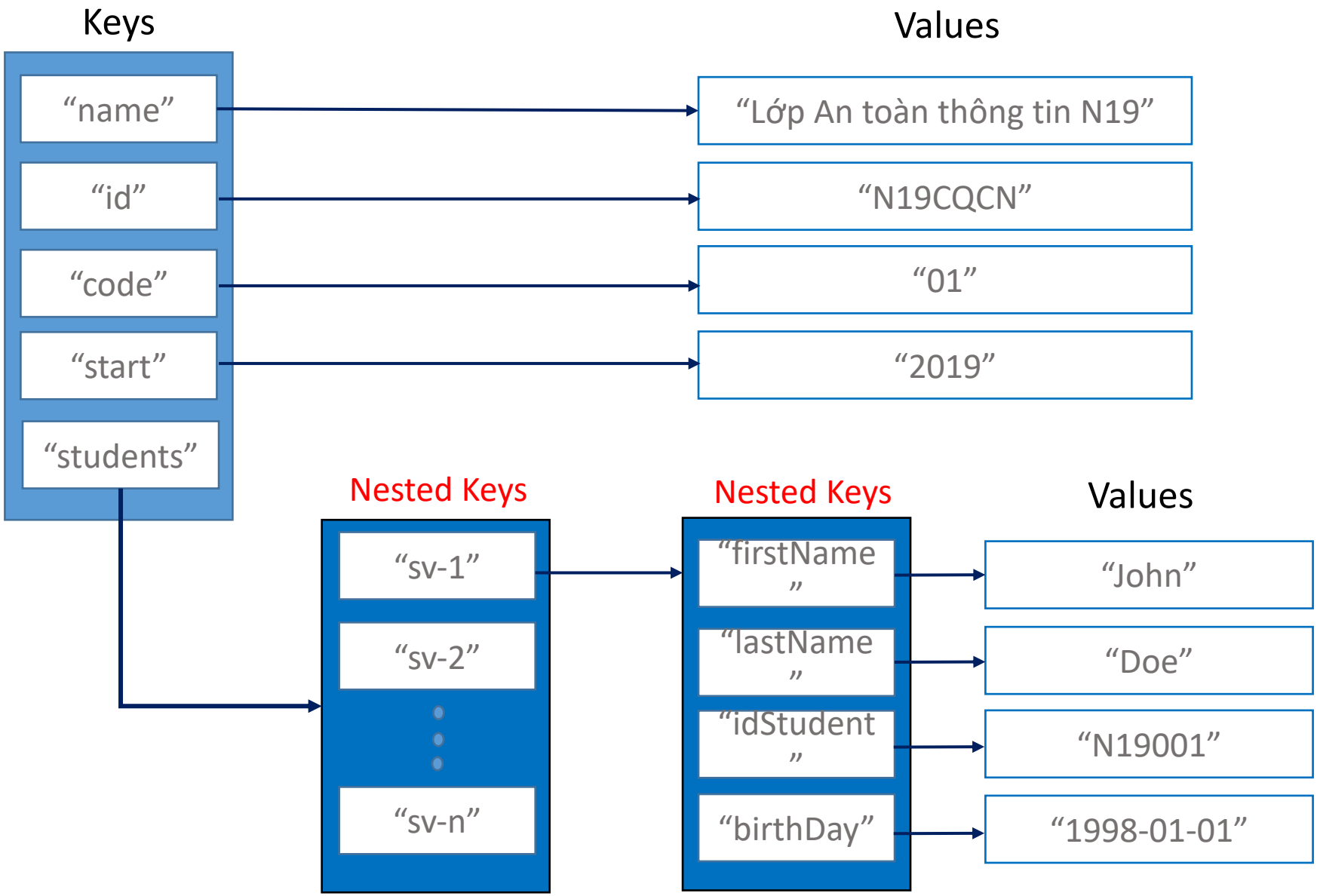
- Write Python code in terminal or script file to count the list of words that are longer than n from a given list of words.

```
>>> Lword = ["abc", "defg", "ghij", "klmno"]
>>> count = 0
>>> n = 3
>>> for x in Lword:
...         if len(x)>n:
...                 count = count+1
…
>>> count
1
```

11. Python - Dictionary

- ***Dictionaries*** are one of the most flexible built-in data types in Python.
- Lists as ordered collections of objects, you can think of dictionaries as unordered collections;
- Items are stored and fetched by ***key***, instead of by positional offset.

- Python dictionaries are:
  - *Accessed by key, not offset position*
  - *Unordered collections of arbitrary objects*
  - *Variable-length, heterogeneous, and arbitrarily nestable*
  - *Of the category "mutable mapping"*
  - *Tables of object references (hash tables)*


Keys
Values
“name”
“Lớp An toàn thông tin N19”
“id”
“N19CQCN”
“code”
“01”
“start”
“2019”
“students”
Nested Keys
Nested Keys
Values
“sv-1”
“sv-2”
“sv-n”
“firstName”
“lastName”
“idStudent”
“birthDay”
“John”
“Doe”
“N19001”
“1998-01-01”

Difference between List and Dictionary:

List	Dictionary
List is a collection of index values pairs as that of array in c++.	Dictionary is a hashed structure of **key and value** pairs.
List is created by placing elements in **[ ]** separated by commas “,”	Dictionary is created by placing elements in **{ }** as “key”:”value”, each key value pair is separated by commas “, “
The indices of list are integers starting from 0.	The keys of dictionary can be of any data type.
The elements are accessed via indices.	The elements are accessed via key-values.
The order of the elements entered are maintained.	There is no guarantee for maintaining order.

Common dictionary literals and operations (T.8.2, p.260-261)

Operation	Interpretation
D = {}	Empty dictionary
D = {'name': 'Bob', 'age': 40}	Two-item dictionary
E = {'cto': {'name': 'Bob', 'age': 40}}	Nesting
D = dict(name='Bob', age=40)	Alternative construction techniques: keywords, key/value pairs, zipped key/value pairs, key lists
D = dict([('name', 'Bob'), ('age', 40)])	
D = dict(zip(keyslist, valueslist))	
D = dict.fromkeys(['name', 'age'])	
D['name']	Indexing by key
E['cto']['age']	

Common dictionary literals and operations (T.8.2, p.260-261)

Operation	Interpretation
'age' in D	Membership: key present test
D.keys()	Methods: all keys,
D.values()	all values,
D.items()	all key+value tuples,
D.copy()	copy (top-level),
D.clear()	clear (remove all items),
D.update(D2)	merge by keys,
D.get(key, default?)	fetch by key, if absent default (or None),
D.pop(key, default?)	remove by key, if absent default (or error)

Common dictionary literals and operations (T.8.2, p.260-261)

Operation	Interpretation
D.setdefault(key, default?)	
D.popitem()	remove/return any (key, value) pair; etc.
len(D)	Length: number of stored entries
D[key] = 42	Adding keys, changing key values
del D[key]	Deleting entries by key
list(D.keys())	Dictionary views (Python 3.X)
D1.keys() & D2.keys()	
D.viewkeys(), D.viewvalues()	Dictionary views (Python 2.7)
D = {x: x*2 for x in range(10)}	Dictionary comprehensions (Python 3.X, 2.7)

Basic Dictionary Operations

- You create dictionaries with literals and store and access items by key with indexing:

```
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}          # Make a dictionary

>>> D['spam']                                     # Fetch a value by key

2

>>> D

{'eggs': 3, 'spam': 2, 'ham': 1}
```

- The built-in len function works on dictionaries, too; it returns the number of items stored in the dictionary or, equivalently, the length of its keys list. The dictionary in membership operator allows you to test for key existence, and the keys method returns all the keys in the dictionary.

```
>>> len(D)                          # Number of entries in dictionary
3
>>> 'ham' in D                      # Key membership test alternative
True
>>> list(D.keys())                  # Create a new list of D's keys
['eggs', 'spam', 'ham']
```

Changing Dictionaries in Place

- Dictionaries, like lists, are mutable, so you can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry.
- The del statement works here, too; it deletes the entry associated with the key specified as an index.

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}

>>> D['ham'] = ['grill', 'bake', 'fry']          # Change entry (value=list)
>>> D
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> del D['eggs']                               # Delete entry
>>> D
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}

>>> D['brunch'] = 'Bacon'                       # Add new entry
>>> D
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

More Dictionary Methods

- Dictionary methods provide a variety of type-specific tools
- The dictionary values and items methods return all of the dictionary's values and (*key*,*value*) pair tuples, respectively;
- Along with keys, these are useful in loops that need to step through dictionary entries one by one.

```
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> list(D.values())
[3, 2, 1]
>>> list(D.items())
[('eggs', 3), ('spam', 2), ('ham', 1)]

>>> D.get('spam')                    # A key that is there
2
>>> print(D.get('toast'))            # A key that is missing
None
>>> D.get('toast', 88)
88
```

- The update method provides something similar to concatenation for dictionaries, though it has nothing to do with left-to-right ordering.
- It *merges* the keys and values of one dictionary into another, blindly overwriting values of the same key if there's a clash.

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}

>>> D2 = {'toast':4, 'muffin':5}          # Lots of delicious scrambled order here
>>> D.update(D2)
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
```

Pop method

- The dictionary **pop** method deletes a key from a dictionary and returns the value it had.
- It’s similar to the list pop method, but it takes a key instead of an optional position.

pop a dictionary by key

```
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
>>> D.pop('muffin')
5
>>> D.pop('toast')                    # Delete and return from a key
4
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
```

pop a list by position

```
>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()                    # Delete and return from the end
'dd'
>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)                   # Delete from a specific position
'bb'
>>> L
['aa', 'cc']
```

Example: Monty Python movie database

- Creates a simple in-memory Monty Python movie database, as a table that maps movie release date *years* (the keys) to movie *titles* (the values). As coded, you fetch movie names by indexing on release year strings:

```
>>> table = {'1975': 'Holy Grail',                # Key: Value
...          '1979': 'Life of Brian',
...          '1983': 'The Meaning of Life'}
>>>
>>> year = '1983'
>>> movie = table[year]                           # dictionary[Key] => Value
>>> movie
'The Meaning of Life'
```

```
>>> for year in table:                    # Same as: for year in table.keys()
...         print(year + '\t' + table[year])
...
1979    Life of Brian
1975    Holy Grail
1983    The Meaning of Life
```

Summary

- Creating Python Dictionary
- Accessing Elements from Dictionary
- Changing and Adding Dictionary elements
- Removing elements from Dictionary
- Python Dictionary Methods
- Python Dictionary Comprehension

Creating Python Dictionary

- Creating a dictionary is as simple as placing items inside curly braces **{}** separated by **commas**.
- An item has a key and a corresponding value that is expressed as a pair (**key: value**).
- While the **values** can be of **any data type** and can repeat, **keys must be of immutable type** (string, number or tuple with immutable elements) and **must be unique**.
- We can also create a dictionary using the built-in **dict()** function.

Accessing Elements from Dictionary

- While indexing is used with other data types to access values, a **dictionary** uses **keys**. Keys can be used either **inside square brackets []** or with the **get()** method.
- If we use the square **brackets [], KeyError** is raised in case a key is not found in the dictionary. On the other hand, the **get()** method returns **None** if the key is not found.

Changing and Adding Dictionary elements

- Dictionaries are **mutable**.

  We can add new items or change the value of existing items using an assignment operator.

- If the key is **already** present, then the existing value gets **updated**. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

Removing elements from Dictionary

- We can remove a particular item in a dictionary by using the **pop()** method. This method removes an item with the **provided key and returns the value**.
- The **popitem()** method can be used to remove and return an **arbitrary (key, value)** item pair from the dictionary.
- All the items can be removed at once, using the **clear()** method.
- We can also use the **del** keyword to remove individual items or the entire dictionary itself.

Exercise 1.0

- Write a Python program to convert them into a dictionary in a way that item from list1 is the key and item from list2 is the value

```
>>> keys = ['eggs', 'one', 2]
>>> values = ['qua_trung’ , 1, 'two']
```

Expected output:

```
{'eggs': 'qua_trung', 'one': 1, 2: 'two'}
```

- **Solution** 1: The zip() function and a dict() constructor
- **Solution 2**: Using a loop and update() method of a dictionary

Exercise 1.0

- **Solution** 1: The zip() function and a dict() constructor

```
>>> keys = ['eggs', 'one', 2]
>>> values = ['qua_trung’ , 1, 'two']
>>> resDict = dict(zip(keys, values))
>>> print(resDict)
```

- **Solution 2**: Using a loop and update() method of a dictionary

```
>>> keys = ['eggs', 'one', 2]
>>> values = ['qua_trung’ , 1, 'two']
>>> resDict = dict()                                  # empty dictionary
>>> for i in range(len(keys)):
>>>       resDict.update({keys[i]: values[i]})
>>> print(resDict)
```

Exercise 2.0

Print the value of key “history” from the below dict

```
>>> sampleDict = {
...         "class": {
…                 "student": {
…                         "name": "Mike",
…                         "marks": {
…                                 "physics": 70,
…                                 "history": 80
...                         }
…                 }
…             }
…}
```

Exercise 3.0

- Write a Python program to create a new dictionary by extracting the mentioned keys from the below dictionary.

```
>>> sample_dict = {
...     "name": "Kelly",
...     "age": 25,
...     "salary": 8000,
...     "city": "New york"}
>>>
>>> # Keys to extract
... keys = ["name", "salary"]
```

Expected output:

```
{'name': 'Kelly', 'salary': 8000}
```