Computing (ES 112)

Yogesh K. Meena
Shouvick Mondal

August 2024

Computer Science & Engineering
IIT Gandhinagar

Dictionaries

Collection: multiple items together

https://www.clarehall.cam.ac.uk/bellcollection/

What is Not a “Collection”? (Recap)

- Most of our variables have one value in them - when we put a new value in the variable, the old value is overwritten

```
$ python3
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on
 linux Type "help", "copyright", "credits" or "license" for more
 information.
>>> x=2
>>> x=4
>>> print(x)
4
```

List as a Collection (Recap)

- A collection allows us to put many values in a single “variable”
- A collection is nice because we can carry all many values (even of different types) around in one convenient package.

```
>>> x = 2
>>> x = 4
>>> print(x)
4
>>> x = [1, 2.35, True, 'A', 5]
print(x)
>>> [1, 2.35, True, 'A', 5]
```

The indexing is exactly like strings. It can even go negative!

```
>>> print(x[0])
>>> print(x[-1])
5
x=[[],[]]
>>> print(len(x))
2
```

Why 2? Should it be zero?

Collections: list, tuple, dictionary

	List	Tuple	Dictionary
Representation	`[1, 2, 3]`	`(1, 2, 3)`	`{'a':1, 'b':2, 'c':3}`
Creation	`list()`, or `[]`	`tuple()`, or `()`	`{}`, or `dict()`
Ordered	Yes	Yes	Yes, since Python 3.7
Mutable	Yes	No	Yes
Homogeneous	No	No	No
Duplicates	Allowed	Allowed	Not Allowed. On duplication attempt, **ONLY the last occurrence remains**.
Index operation []	Valid integers	Valid integers	Only keys by name.
Slice operation :	Allowed	Allowed	Not Allowed

The notion of key-value pair is inherent in dictionaries

Negative indexing such as -1,-2,.. for dictionaries?

Dictionary: Lookup, Read, Write (Python <3.7)

- Lists index their entries based on the position in the list
- But we index the things we put in the dictionary with a 'lookup tag' (string)
- Read operation: the key must be present.
- Write operation: creates or overwrites value.

```
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print(purse)
{'money': 12, 'tissues': 75, 'candy': 3}
>>> print(purse['candy'])
3
>>> purse['candy'] = purse['candy'] + 2
>>> print(purse)
{'money': 12, 'tissues': 75, 'candy': 5}
```

Dictionary: Lookup, Read, Write (Python >=3.7)

- Lists index their entries based on the position in the list
- But we index the things we put in the dictionary with a 'lookup tag' (string)
- Read operation: the key must be present.
- Write operation: creates or overwrites value.

```
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print(purse)
{'money': 12, 'candy': 3, 'tissues': 75}
>>> print(purse['candy'])
3
>>> purse['candy'] = purse['candy'] + 2
>>> print(purse)
{'money': 12, 'candy': 5, 'tissues': 75}
```

Semantic fix!

Insertion: List versus Dictionary

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'age': 21, 'course': 182}
>>> ddd['age'] = 23
>>> print(ddd)
{'age': 23, 'course': 182}
```

Key	Value
[0]	23
[1]	183

Key	Value
['age']	23
['course']	183

Deletion: List versus Dictionary

```
>>> L=[10,9,8,7]
>>> L.pop()
7
>>> print(L)
[10, 9, 8]
>>> L=[10,9,8,7]
>>> L.remove(8)
>>> print(L)
[10, 9, 7]
```

parameter-free

parameter-bound

```
>>> D={'a':10,'b':9,'c':-1,'c':8,'d':9}
>>> print(D)
{'a': 10, 'b': 9, 'c': 8, 'd': 9}
>>> D.popitem()
('d', 9)
>>> print(D)
{'a': 10, 'b': 9, 'c': 8}
>>> D={'a':10,'b':9,'c':-1,'c':8,'d':9}
>>> D.pop('c')
8
>>> print(D)
{'a': 10, 'b': 9, 'd': 9}
```

Last occurrence of c remains

The parameter-free pop operation removes the item that was added most recently. Last-In-First-Out (LIFO)

The parameter-bound pop operation deletes the specified item from the:

- list (first occurrence)
- dictionary (item with the key)

Dictionary Tracebacks

- It is an error to reference a key which is not in the dictionary
- We can use the `in` operator to see if a key is in the dictionary

```
>>> ddd = {'age': 23, 'course': 182}
>>> print(ddd['gender'])
Traceback (most recent call last):
  File "", line 1, in ...
KeyError: 'gender'
```

```
>>> 'gender' in ddd
False
>>> 'age' in ddd
True
```

The `in` goes through the `keys` ONLY, not the values!

Referencing Non-existent Keys without Error

- We can do this using the `get()` function for dictionaries.

- arg #1: key
- arg #2: default value if key not found

```
>>> ddd = {'age': 23, 'course': 182}
>>> val = ddd.get('gender',None)
>>> print(val)
None
>>> val = ddd.get('age',None)
>>> print(val)
23
```

Default value if key does not exist (and no Traceback)

Retrieving Lists of Keys and Values

We can get a list of keys, values, or items (both) from a dictionary

List of tuples
[('a', 1),
('b', 2),
('c', 3),
('d', 4)]

```
>>> D={'a':1,'b':2,'c':3,'d':4}
>>> print(D)
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> print(D.items())
dict_items([('a', 1), ('b', 2), ('c', 3), ('d', 4)])
>>> print(D.keys())
dict_keys(['a', 'b', 'c', 'd'])
>>> print(D.values())
dict_values([1, 2, 3, 4])
>>> print(list(D))
['a', 'b', 'c', 'd']
>>> print(list(D.items()))
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
>>> print(list(D.keys()))
['a', 'b', 'c', 'd']
>>> print(list(D.values()))
[1, 2, 3, 4]
```

Definite Loops and Dictionaries

We can write a for loop that goes through all the entries in a dictionary - actually it goes through all of the keys in the dictionary and looks up the values

```
>>> D={'a':1,'b':2,'c':3,'d':4}
>>> for i in D:
...     print(D[i])
...
1
2
3
4
>>> for i,j in D.items():
...     print(i,j)
...
a 1
b 2
c 3
d 4
```

One iteration variable

Two iteration variables

Programs for Python… (Recap)

the clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car

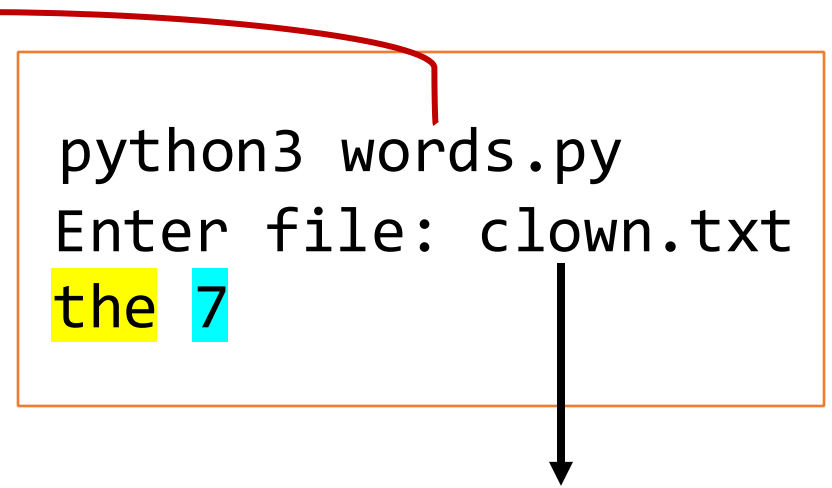
Which is the most frequently occurring word here?

Image: https://www.flickr.com/photos/allan_harris/4908070612/ Attribution-NoDerivs 2.0 Generic (CC BY-ND 2.0)

Count the most frequent word v1 (Recap)

```
name = input('Enter file:')
handle = open(name)
counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1
bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```



Tuples

Collections: list, tuple, dictionary

	List	Tuple	Dictionary
Representation	`[1, 2, 3]`	`(1, 2, 3)`	`{'a':1, 'b':2, 'c':3}`
Creation	`list()`, or `[]`	`tuple()`, or `()`	`{}`, or `dict()`
Ordered	Yes	Yes	Yes, since Python 3.7
Mutable	Yes	No	Yes
Homogeneous	No	No	No
Duplicates	Allowed	Allowed	Not Allowed
Index operation []	Valid integers	Valid integers	Only keys by name
Slice operation :	Allowed	Allowed	Not Allowed
Concatenation	**+ (plus)**	**+ (plus)**	**| (pipe)**
Basic comparison	**<, >, <=, >=, ==, != lexicographical**	**<, >, <=, >=, ==, != lexicographical**	**==, != unordered**

Tuples are like Lists

- Tuples are another kind of sequence that functions much like a list
- They have elements which are indexed starting at 0
- Even negative indexing is also possible. Hence, slicing is also possible.

```
>>> x=('I','A','N','R')
>>> print(x[2])
N
>>> print(x[-1])
R
>>> y=(1,2,3)
>>> print(y)
(1, 2, 3)
>>> print(y[1:])
(2, 3)
>>> print(y[1:2])
(2,)
>>> print(y[1:1])
()
```

```
>>> x=['I','A','N','R']
>>> print(x[2])
N
>>> print(x[-1])
R
>>> y=[1,2,3]
>>> print(y)
[1, 2, 3]
>>> print(y[1:])
[2, 3]
>>> print(y[1:2])
[2]
>>> print(y[1:1])
[]
```

Iteration? Yes

By def'n, a tuple must be at least be a couple so, the comma after performing slicing remains, the second component is blank

Tuples are like Lists but immutable

Unlike a list, once you create a tuple, it is read-only - like a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
[9, 8, 6]
```

```
>>> x = 'ABC'
>>> x[2] = 'D'
Traceback (most recent
call last): File "",
line 1, in TypeError:
'str' object does not
support item assignment
```

```
>>> z = (5,4,3)
>>> z[2] = 0
Traceback (most recent
call last): File "",
line 1, in TypeError:
'tuple' object does not
support item assignment
```

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- Hence, in our program when we are making “temporary variables” we prefer tuples over lists

Things (not) to do with Tuples

Because tuples are read-only, you cannot perform the following operations as they would mutate the tuple.

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no
attribute 'sort'
>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no
attribute 'append'
>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no
attribute 'reverse'
```

```
>>> x = (3, 2, 1)
>>>
>>> x = tuple(sorted(x))
>>> print(x)
(1, 2, 3)
>>>
>>> x += (1, 2)
>>> print(x)
(1, 2, 3, 1, 2)
>>>
>>> x=tuple(sorted(x,reverse=True))
>>> print(x)
(3, 2, 2, 1, 1)
```

Assignment, Packing, and Unpacking

- We can also put a tuple on the left-hand side of an assignment statement
- We can even omit the parentheses

```
>>> (x,y) = (1,'India')
>>> print(y)
India
>>> a,b = (2,3)
>>> print(a)
2
>>> print(b)
3
```

- When we create a tuple, we normally assign values to it. This is called "packing" a tuple.

```
>>> fruits = ("apple", "banana", "cherry")
>>> print(fruits)
('apple', 'banana', 'cherry')
```

- We are also allowed to extract the values back into variables. This is called "unpacking"

```
>>> green,yellow,red = fruits
>>> print(green)
apple
```

The number of variables on the L.H.S must be sufficient enough, otherwise ValueError happens and program crashes

Concatenation: List, Tuple, Dictionary

List

```
>>> a = [1,2,3]
>>> b = a + [5]
>>> print(b)
[1, 2, 3, 5]
>>>
>>> c = [1,5] + [3,4]
>>> print(c)
[1, 5, 3, 4]
>>>
>>> c += c
>>> print(c)
[1, 5, 3, 4, 1, 5, 3, 4]
```

Tuple

```
>>> a = (1,2,3)
>>> b = a + (5,)
>>> print(b)
(1, 2, 3, 5)
>>>
>>> c = (1,5) + (3,4)
>>> print(c)
(1, 5, 3, 4)
>>>
>>> c += c
>>> print(c)
(1, 5, 3, 4, 1, 5, 3, 4)
```

Dictionary

```
>>> D = {'a':1,'b':2,'c':3}
>>> E = D | {'d':5}
>>> print(E)
{'a': 1, 'b': 2, 'c': 3, 'd': 5}
>>>
>>> F={'a':2,'b':3}|{'e':6,'k':1}
>>> print(F)
{'a': 2, 'b': 3, 'e': 6, 'k': 1}
>>>
>>> D |= {'b':8}
>>> print(D)
{'a': 1, 'b': 8, 'c': 3}
```

On duplication attempt, **ONLY the last occurrence remains**.

Comparision Operators: List, Tuple, Dictionary

List

```
>>> [1,2,3] > [1,3,2]
False
>>> [1,2,3] == [1,3,2]
False
>>> [1,2,3] == [1,2,3]
True
>>> [1,5] < [1,5,1]
True
>>> [1,5,2] < [1,5,1]
False
>>> [1,1,5,] == [1,5,1]
False
```

Tuple

```
>>> (1,2,3) > (1,3,2)
False
>>> (1,2,3) == (1,3,2)
False
>>> (1,2,3) == (1,2,3)
True
>>> (1,5) < (1,5,1)
True
>>> (1,5,2) < (1,5,1)
False
>>> (1,1,5) == (1,5,1)
False
```

Dictionary

```
>>> {'a':1,'b':2,'c':3} ==
{'a':1,'c':3,'b':2}
True
>>>
>>>
>>> {'a':1,'b':5} <
{'a':1,'b':5,'c':1}
Traceback (most recent call
last): File "", line 1, in
TypeError: '<' not supported
between instances of 'dict' and
'dict'
```

The comparison operators work with lists and tuples in the same capacity as it works with string comparisons, i.e., lexicographically performed

Retrieving Lists of (keys, values) from a dict...

We can get a list of keys, values, or items (both) from a dictionary

List of tuples
[('a', 1),
('b', 2),
('c', 3),
('d', 4)]

```
>>> D={'a':1,'b':2,'c':3,'d':4}
>>> print(D)
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> print(D.items())
dict_items([('a', 1), ('b', 2), ('c', 3), ('d', 4)])
>>> print(D.keys())
dict_keys(['a', 'b', 'c', 'd'])
>>> print(D.values())
dict_values([1, 2, 3, 4])
>>> print(list(D))
['a', 'b', 'c', 'd']
>>> print(list(D.items()))
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
>>> print(list(D.keys()))
['a', 'b', 'c', 'd']
>>> print(list(D.values()))
[1, 2, 3, 4]
```

Programs for Python... (Recap)

the clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car

Which is the most frequently occurring word here?

Image: https://www.flickr.com/photos/allan_harris/4908070612/ Attribution-NoDerivs 2.0 Generic (CC BY-ND 2.0)

Count the most frequent word v2

```
fname = input('Enter a file name:')
fp = open(fname)
d = {}
while True:
  i = fp.readline()
  if len(i) == 0:
    break
  else:
    k = i.split()
    if len(k) != 0:
      for j in k:
        d[j] = d.get(j,0) + 1

z = [(x,y) for y,x in d.items()]
freq,high = sorted(z,reverse=True)[0]
print(high,freq)
fp.close()
```

What are we doing here?

- Comprehension
- Reverse sorting
- Double iteration variable y,x

```
python3 words.py
Enter file: clown.txt
the 7
```

the clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car

Count the most frequent word v2

Writing programs or programming is a very creative and rewarding activity You can write programs for many reasons ranging from making your living to solving a difficult data analysis problem to having fun to helping someone else solve a problem This book assumes that {\em everyone} needs to know how to program and that once you know how to program, you will figure out what you want to do with your newfound skills

We are surrounded in our daily lives with computers ranging from laptops to cell phones We can think of these computers as our personal assistants who can take care of many things on our behalf The hardware in our current-day computers is essentially built to continuously ask us the question What would you like me to do next

Our computers are fast and have vasts amounts of memory and could be very helpful to us if we only knew the language to speak to explain to the computer what we would like it to do next If we knew this language we could tell the computer to do tasks on our behalf that were reptitive Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing

```
python words.py
Enter file: words.txt
to 16
```

Dictionary State During Final Search

```
[(1, 'Writing'), (2, 'programs'), (1, 'or'), (1, 'programming'), (2, 'is'), (3, 'a'), (2, 'very'),
(1, 'creative'), (5, 'and'), (1, 'rewarding'), (1, 'activity'), (1, 'You'), (4, 'can'), (1,
'write'), (1, 'for'), (2, 'many'), (1, 'reasons'), (2, 'ranging'), (2, 'from'), (1, 'making'), (2,
'your'), (1, 'living'), (16, 'to'), (1, 'solving'), (1, 'difficult'), (1, 'data'), (1,
'analysis'), (2, 'problem'), (1, 'having'), (1, 'fun'), (1, 'helping'), (1, 'someone'), (1,
'else'), (1, 'solve'), (1, 'This'), (1, 'book'), (1, 'assumes'), (4, 'that'), (1, '{\\em'), (1,
'everyone}'), (1, 'needs'), (2, 'know'), (2, 'how'), (1, 'program'), (1, 'once'), (4, 'you'), (1,
'program,'), (1, 'will'), (1, 'figure'), (1, 'out'), (2, 'what'), (1, 'want'), (5, 'do'), (2,
'with'), (1, 'newfound'), (1, 'skills'), (2, 'We'), (3, 'are'), (1, 'surrounded'), (2, 'in'), (5,
'our'), (1, 'daily'), (1, 'lives'), (5, 'computers'), (1, 'laptops'), (1, 'cell'), (1, 'phones'),
(1, 'think'), (5, 'of'), (1, 'these'), (1, 'as'), (1, 'personal'), (1, 'assistants'), (1, 'who'),
(1, 'take'), (1, 'care'), (3, 'things'), (2, 'on'), (2, 'behalf'), (1, 'The'), (1, 'hardware'),
(1, 'current-day'), (1, 'essentially'), (1, 'built'), (1, 'continuously'), (1, 'ask'), (2, 'us'),
(6, 'the'), (1, 'question'), (1, 'What'), (2, 'would'), (2, 'like'), (1, 'me'), (2, 'next'), (1,
'Our'), (1, 'fast'), (1, 'have'), (1, 'vasts'), (1, 'amounts'), (1, 'memory'), (2, 'could'), (1,
'be'), (1, 'helpful'), (1, 'if'), (5, 'we'), (1, 'only'), (2, 'knew'), (2, 'language'), (1,
'speak'), (1, 'explain'), (2, 'computer'), (1, 'it'), (1, 'If'), (1, 'this'), (1, 'tell'), (1,
'tasks'), (1, 'were'), (1, 'reptitive'), (1, 'Interestingly,'), (2, 'kinds'), (1, 'best'), (1,
'often'), (1, 'humans'), (1, 'find'), (1, 'boring'), (1, 'mind-numbing')]
```

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Contributors 2024 - Yogesh K. Meena and Shouvick Mondal, IIT Gandhinagar