

MODULE-2

Lists, Dictionaries and Structuring Data

Dr. Sandeep Bhat

Professor, CSE Dept
SIT, Mangaluru

Lists

The List Data Type

- A list is a value that contains multiple values in an ordered sequence.
- The term list value refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value.
- A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].

Cont..

- ▶ Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, [].
- ▶ Values inside the list are also called *items*. Items are separated with commas (that is, they are *comma-delimited*).

Eg.

```
>>> [1, 2, 3]
[1, 2, 3]

>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']

>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]

❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam
['cat', 'bat', 'rat', 'elephant']
```

Cont..

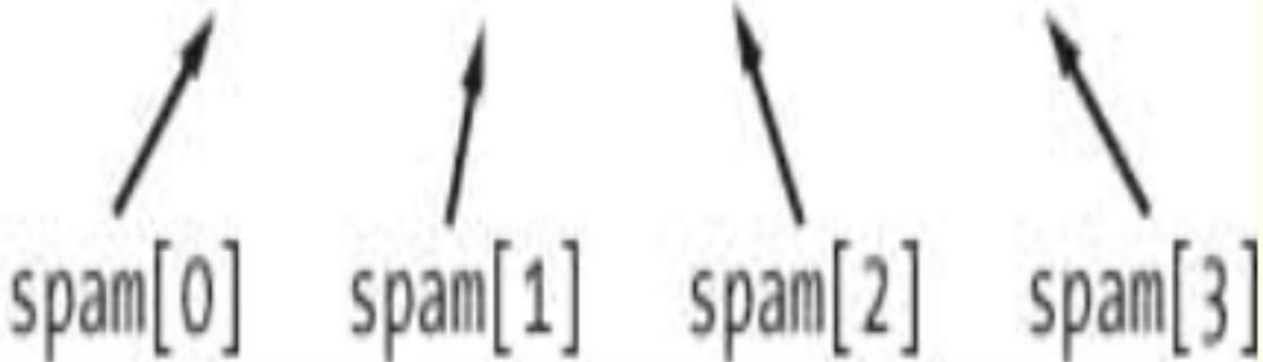
- The spam variable is still assigned only one value: the list value.
- But the list value itself contains other values.
- The value `[]` is an empty list that contains no values, similar to `"`, the empty string.

Getting Individual Values in a List with Indexes

- Say you have the list `['cat', 'bat', 'rat', 'elephant']` stored in a variable named `spam`.
- The Python code `spam[0]` would evaluate to `'cat'`, and `spam[1]` would evaluate to `'bat'`, and so on.
- The integer inside the square brackets that follows the list is called an *index*.
- The first value in the list is at index 0, the second value is at index 1, the third value is at index 2, and so on.

Cont..

```
spam = ["cat", "bat", "rat", "elephant"]
```



The diagram illustrates the indexing of the list `spam`. It shows the list `spam = ["cat", "bat", "rat", "elephant"]` at the top. Below it, the indices `spam[0]`, `spam[1]`, `spam[2]`, and `spam[3]` are displayed. Arrows point from each index to its corresponding element in the list: `spam[0]` points to "cat", `spam[1]` points to "bat", `spam[2]` points to "rat", and `spam[3]` points to "elephant".

- For example, enter the following expressions into the interactive shell. Start by assigning a list to the variable `spam`.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[0]
```

```
'cat'
```

```
>>> spam[1]
```

```
'bat'
```

```
>>> spam[2]
```

```
'rat'
```


Cont..

```
>>> spam[3]
```

```
'elephant'
```

```
>>> ['cat', 'bat', 'rat', 'elephant'][3]
```

```
'elephant'
```

```
❶ >>> 'Hello, ' + spam[0]
```

```
❷ 'Hello, cat'
```

```
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
```

```
'The bat ate the cat.'
```

Negative Indexes

- While indexes start at 0 and go up, you can also use negative integers for the index.
- The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on.

Cont..

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
  
>>> spam[-1]  
  
'elephant'  
  
>>> spam[-3]  
  
'bat'  
  
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'  
  
'The elephant is afraid of the bat.'
```

Getting a List from Another List with Slices

- ▶ Just as an index can get a single value from a list, a slice can get several values from a list, in the form of a new list.
- ▶ A slice is typed between square brackets, like an index, but it has two integers separated by a colon.
- ▶ Notice the difference between indexes and slices.

- `spam[2]` is a list with an index (one integer).
- `spam[1:4]` is a list with a slice (two integers).

Cont..

- In a slice, the first integer is the index where the slice starts.
- The second integer is the index where the slice ends.
- A slice goes up to, but will not include, the value at the second index.
- A slice evaluates to a new list value.

Eg.1

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
  
>>> spam[0:4]  
['cat', 'bat', 'rat', 'elephant']  
  
>>> spam[1:3]  
['bat', 'rat']  
  
>>> spam[0:-1]  
['cat', 'bat', 'rat']
```

Eg. 2

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
>>> spam[:2]  
['cat', 'bat']  
>>> spam[1:]  
['bat', 'rat', 'elephant']  
>>> spam[:]  
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with the len() Function

- ▶ The len() function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value.

```
>>> spam = ['cat', 'dog', 'moose']
```

```
>>> len(spam)
```

```
3
```


Changing Values in a List with Indexes

- Normally, a variable name goes on the left side of an assignment statement, like **spam = 42**.
- However, you can also use an index of a list to change the value at that index.
- For example, **spam[1] = 'aardvark'** means “Assign the value at index 1 in the list spam to the string 'aardvark'.”

Eg.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
  
>>> spam[1] = 'aardvark'  
  
>>> spam  
  
['cat', 'aardvark', 'rat', 'elephant']  
  
>>> spam[2] = spam[1]  
  
>>> spam
```

Cont..

```
['cat', 'aardvark', 'aardvark', 'elephant']
```

```
>>> spam[-1] = 12345
```

```
>>> spam
```

```
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

- Lists can be concatenated and replicated just like strings.
- The + operator combines two lists to create a new list value and the * operator can be used with a list and an integer value to replicate the list.

Cont..

```
>>> [1, 2, 3] + ['A', 'B', 'C']  
[1, 2, 3, 'A', 'B', 'C']  
  
>>> ['X', 'Y', 'Z'] * 3  
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']  
  
>>> spam = [1, 2, 3]  
  
>>> spam = spam + ['A', 'B', 'C']  
  
>>> spam  
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

- The del statement will delete values at an index in a list.
- All of the values in the list after the deleted value will be moved up one index.

Cont..

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
  
>>> del spam[2]  
  
>>> spam  
  
['cat', 'bat', 'elephant']  
  
>>> del spam[2]  
  
>>> spam  
  
['cat', 'bat']
```

WORKING WITH LISTS

- ▶ When you first begin writing programs, it's tempting to create many individual variables to store a group of similar values.
- ▶ For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

Cont..

```
catName1 = 'Zophie'  
catName2 = 'Pooka'  
catName3 = 'Simon'  
catName4 = 'Lady Macbeth'  
catName5 = 'Fat-tail'  
catName6 = 'Miss Cleo'
```

Cont..

- It turns out that this is a bad way to write code. (Also, I don't actually own this many cats, I swear.)
- For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables.
- These types of programs also have a lot of duplicate or nearly identical code in them.

Eg.

```
print('Enter the name of cat 1:')  
catName1 = input()  
print('Enter the name of cat 2:')  
catName2 = input()  
print('Enter the name of cat 3:')  
catName3 = input()  
print('Enter the name of cat 4:')  
catName4 = input()
```

Contd..

```
print('Enter the name of cat 5:')  
  
catName5 = input()  
  
print('Enter the name of cat 6:')  
  
catName6 = input()  
  
print('The cat names are:')  
  
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +  
catName5 + ' ' + catName6)
```

Drawback of the program:

- **Instead of using multiple, repetitive variables, you can use a single variable that contains a list value.**
- **For example, here's a new and improved version of the program....**

improved version of the program....

```
catNames = []  
  
while True:  
    print('Enter the name of cat ' + str(len(catNames) + 1) +  
          ' (Or enter nothing to stop.):')  
    name = input()  
    if name == '':
```

Cont..

```
break

catNames = catNames + [name] # list concatenation

print('The cat names are:')

for name in catNames:

    print(' ' + name)
```

When you run this program, the output will look something like this:

```
Enter the name of cat 1 (Or enter nothing to stop.):
```

```
Zophie
```

```
Enter the name of cat 2 (Or enter nothing to stop.):
```

```
Pooka
```

```
Enter the name of cat 3 (Or enter nothing to stop.):
```

```
Simon
```

```
Enter the name of cat 4 (Or enter nothing to stop.):
```

```
Lady Macbeth
```


Cont..

Lady Macbeth

Enter the name of cat 5 (Or enter nothing to stop.):

Fat-tail

Enter the name of cat 6 (Or enter nothing to stop.):

Miss Cleo

Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:

Zophie

Pooka

Simon

Lady Macbeth

Fat-tail

Miss Cleo

Dr. S. Madan Bhat on Mon Jul 31

7/8/2023

Using for Loops with Lists

eg.

```
for i in range(4):  
    print(i)
```

the output of this program would be as follows:

0

1

2

3

Cont..

- This is because the return value from `range(4)` is a sequence value that Python considers similar to `[0, 1, 2, 3]`.
- The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:  
    print(i)
```

Cont..

- ▶ A common Python technique is to use `range(len(someList))` with a for loop to iterate over the indexes of a list.
- ▶ For example, enter the following into the interactive shell:

Cont..

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']  
>>> for i in range(len(supplies)):  
...     print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

Index 0 in supplies is: pens

Index 1 in supplies is: staplers

Index 2 in supplies is: flamethrowers

Index 3 in supplies is: binders

The Multiple Assignment Trick

- The *multiple assignment trick* (technically called *tuple unpacking*) is a shortcut that lets you assign multiple variables with the values in a list in one line of code.
- So instead of doing this:

Cont..

```
>>> cat = ['fat', 'gray', 'loud']  
>>> size = cat[0]  
>>> color = cat[1]  
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'gray', 'loud']  
>>> size, color, disposition = cat
```

Cont..

- ▶ The number of variables and the length of the list must be exactly equal, or Python will give you a `ValueError`:

```
>>> cat = ['fat', 'gray', 'loud']  
  
>>> size, color, disposition, name = cat  
  
Traceback (most recent call last):  
  
  File "<pyshell#84>", line 1, in <module>  
    size, color, disposition, name = cat  
  
ValueError: not enough values to unpack (expected 4, got 3)
```


Using the enumerate() Function with Lists

- Instead of using the `range(len(someList))` technique with a for loop to obtain the integer index of the items in the list, you can call the `enumerate()` function instead.
- On each iteration of the loop, `enumerate()` will return two values: the index of the item in the list, and the item in the list itself.

- The `enumerate()` function is useful if you need both the item and the item's index in the loop's block.

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']  
>>> for index, item in enumerate(supplies):  
...     print('Index ' + str(index) + ' in supplies is: ' + item)
```

```
Index 0 in supplies is: pens
```

```
Index 1 in supplies is: staplers
```

```
Index 2 in supplies is: flamethrowers
```

```
Index 3 in supplies is: binders
```

AUGMENTED ASSIGNMENT OPERATORS

- ▶ When assigning a value to a variable, you will frequently use the variable itself.
- ▶ For example, after assigning 42 to the variable spam, you would increase the value in spam by 1 with the following code:

Cont..

```
>>> spam = 42
```

```
>>> spam += 1
```

```
>>> spam
```

```
43
```

Cont..

Table 4-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
<code>spam += 1</code>	<code>spam = spam + 1</code>
<code>spam -= 1</code>	<code>spam = spam - 1</code>
<code>spam *= 1</code>	<code>spam = spam * 1</code>
<code>spam /= 1</code>	<code>spam = spam / 1</code>
<code>spam %= 1</code>	<code>spam = spam % 1</code>

METHODS

- A *method* is the same thing as a function, except it is “called on” a value.
- For example, if a list value were stored in `spam`, you would call the `index()` list method on that list like so: `spam.index('hello')`.
- The **method** part comes after the value, separated by a period.

Cont..

- Each data type has its own set of methods.
- The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the index() Method

- List values have an `index()` method that can be passed a value, and if that value exists in the list, the index of the value is returned.
- If the value isn't in the list, then Python produces a `ValueError` error.

Cont..

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
```

```
>>> spam.index('hello')
```

```
0
```

```
>>> spam.index('heyas')
```

```
3
```

```
>>> spam.index('howdy howdy howdy')
```

```
Traceback (most recent call last):
```

```
File "<pyshell#31>", line 1, in <module>
```

```
    spam.index('howdy howdy howdy')
```

```
ValueError: 'howdy howdy howdy' is not in list
```

Adding Values to Lists with the `append()` and `insert()` Methods

- To add new values to a list, use the `append()` and `insert()` methods.
- Enter the following into the interactive shell to call the `append()` method on a list value stored in the variable `spam`:

Cont...

```
>>> spam = ['cat', 'dog', 'bat']  
  
>>> spam.append('moose')  
  
>>> spam  
  
['cat', 'dog', 'bat', 'moose']
```

- The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list.

Cont..

```
>>> spam = ['cat', 'dog', 'bat']  
>>> spam.insert(1, 'chicken')  
>>> spam  
['cat', 'chicken', 'dog', 'bat']
```

Removing Values from Lists with the `remove()` Method

- The `remove()` method is passed the value to be removed from the list it is called on.
- Eg.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
>>> spam.remove('bat')  
>>> spam  
['cat', 'rat', 'elephant']
```

Sorting the Values in a List with the sort() Method

- ▶ Lists of number values or lists of strings can be sorted with the `sort()` method.
- ▶ For example, enter the following into the interactive shell:

Cont..

```
>>> spam = [2, 5, 3.14, 1, -7]
```

```
>>> spam.sort()
```

```
>>> spam
```

```
[-7, 1, 2, 3.14, 5]
```

```
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
```

```
>>> spam.sort()
```

```
>>> spam
```

```
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

EXAMPLE PROGRAM: MAGIC 8 BALL WITH A LIST

- Instead of several lines of nearly identical **elif** statements, you can create a single list that the code works with.

Cont..

```
import random

messages = ['It is certain',
            'It is decidedly so',
            'Yes definitely',
            'Reply hazy try again',
            'Ask again later',
```

Cont..

```
'My reply is no',  
'Outlook not so good',  
'Very doubtful']
```

```
print(messages[random.randint(0, len(messages) - 1)])
```

Cont..

- ▶ Notice the expression you use as the index for messages: `random.randint(0, len(messages) - 1)`.
- ▶ This produces a random number to use for the index, regardless of the size of messages.
- ▶ That is, you'll get a random number between 0 and the value of `len(messages) - 1`.
- ▶ The benefit of this approach is that you can easily add and remove strings to the messages list without changing other lines of code.

SEQUENCE DATA TYPES

- Lists aren't the only data types that represent ordered sequences of values.
- For example, strings and lists are actually similar if you consider a string to be a “**list**” of single text characters.
- The Python sequence data types include lists, strings, range objects returned by **range()**, and tuples

The Tuple Data Type

- The *tuple* data type is almost identical to the list data type, except in two ways.
- First, tuples are typed with parentheses, (and), instead of square brackets, [and].
- For example:

Cont..

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
```

Converting Types with the list() and tuple() Functions

- ▶ Just like how `str(42)` will return `'42'`, the string representation of the integer 42, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them.
- ▶ Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:

Cont..

```
>>> tuple(['cat', 'dog', 5])  
('cat', 'dog', 5)  
  
>>> list(('cat', 'dog', 5))  
['cat', 'dog', 5]  
  
>>> list('hello')  
['h', 'e', 'l', 'l', 'o']
```

Lists to Tuple conversion

```
my_list = [1, 2, 3, 4, 5]  
my_tuple = tuple(my_list)  
print(my_tuple)
```

Output:

SCSS

```
(1, 2, 3, 4, 5)
```

REFERENCES

- The variables “store” strings and integer values.
- However, this explanation is a simplification of what Python is actually doing.
- Technically, variables are storing references to the computer memory locations where the values are stored.

Cont..

```
>>> spam = 42  
  
>>> cheese = spam  
  
>>> spam = 100  
  
>>> spam  
  
100  
  
>>> cheese  
  
42
```

Cont..

- When you assign 42 to the spam variable, you are actually creating the 42 value in the computer's memory and storing a *reference* to it in the spam variable.
- When you copy the value in spam and assign it to the variable cheese, you are actually copying the reference.
- Both the spam and cheese variables refer to the 42 value in the computer's memory.

Cont..

- When you later change the value in `spam` to 100, you're creating a new 100 value and storing a reference to it in `spam`.
- This doesn't affect the value in `cheese`. Integers are *immutable* values that don't change; changing the `spam` variable is actually making it refer to a completely different value in memory.

DICTIONARIES AND STRUCTURING DATA

THE DICTIONARY DATA TYPE

- ▶ Like a list, a *dictionary* is a mutable collection of many values.
- ▶ But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers.
- ▶ Indexes for dictionaries are called *keys*, and a key with its associated value is called a *key-value pair*.

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are 'size', 'color', and 'disposition'. The values for these keys are 'fat', 'gray', and 'loud', respectively. You can access these values through their keys:

Cont..

```
>>> myCat['size']  
  
'fat'  
  
>>> 'My cat has ' + myCat['color'] + ' fur.'  
  
'My cat has gray fur.'
```

Dictionaries vs. Lists

- Unlike lists, items in dictionaries are unordered.
- The first item in a list named spam would be spam[0].
- But there is no “first” item in a dictionary.
- While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

Cont..

```
>>> spam = ['cats', 'dogs', 'moose']
```

```
>>> bacon = ['dogs', 'moose', 'cats']
```

```
>>> spam == bacon
```

False

```
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
```

```
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
```

```
>>> eggs == ham
```

True

Cont..

- Because dictionaries are not ordered, they can't be sliced like lists.
- Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message.
- Enter the following into the interactive shell, and notice the error message that shows up because there is no 'color' key

The keys(), values(), and items() Methods

- There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`.
- The values returned by these methods are not true lists: they cannot be modified and do not have an `append()` method.
- But these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) can be used in for loops.

Cont..

```
>>> spam = {'color': 'red', 'age': 42}
```

```
>>> for v in spam.values():
```

```
...     print(v)
```

```
red
```

```
42
```

Cont..

```
>>> for k in spam.keys():  
...     print(k)  
  
color  
age  
  
>>> for i in spam.items():  
...     print(i)  
  
('color', 'red')  
('age', 42)
```

Cont..

- ▶ When you use the `keys()`, `values()`, and `items()` methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively.
- ▶ Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value.

Checking Whether a Key or Value Exists in a Dictionary

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
```


Cont..

```
>>> 'color' not in spam.keys()
```

```
True
```

```
>>> 'color' in spam
```

```
False
```

Cont..

- In the previous example, notice that 'color' in spam is essentially a shorter version of writing 'color' in **spam.keys()**.
- This is always the case: if you ever want to check whether a value is (or isn't) a key in the dictionary, you can simply use the in (or not in) keyword with the dictionary value itself.

The get() Method

- ▶ It's tedious to check whether a key exists in a dictionary before accessing that key's value.
- ▶ Fortunately, dictionaries have a **get()** method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Cont..

```
>>> picnicItems = {'apples': 5, 'cups': 2}

>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'

'I am bringing 2 cups.'

>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'

'I am bringing 0 eggs.'
```

Cont..

- Because there is no 'eggs' key in the **picnicItems** dictionary, the default value 0 is returned by the `get()` method.
- Without using **`get()`**, the code would have caused an error message.

The.setdefault() Method

- You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value.

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

Cont..

- The **setdefault()** method offers a way to do this in one line of code.
- The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist.
- If the key does exist, the **setdefault()** method returns the key's value.

PRETTY PRINTING

- If you import the `pprint` module into your programs, you'll have access to the `pprint()` and `pformat()` functions that will "pretty print" a dictionary's values.
- This is helpful when you want a cleaner display of the items in a dictionary than what `print()` provides.

Example

```
import pprint

message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'

count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

Output

```
{' ': 13,  
  ',': 1,  
  '.': 1,  
  'A': 1,  
  'I': 1,  
  --snip--  
  't': 6,  
  'w': 2,  
  'y': 1}
```
