

Module 2

Python Programming Basics

Lists, Dictionaries and Structuring Data

Syllabus:

- **Lists:** The List Data Type, Working with Lists, Augmented Assignment Operators, Methods, Example Program: Magic 8 Ball with a List, List-like Types: Strings and Tuples, References,
- **Dictionaries and Structuring Data:** The Dictionary Data Type, Pretty Printing, Using Data Structures to Model Real-World Things,
- **Textbook 1: Chapters 4 – 5**

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 not the values inside the list value.
- The list value can be stored in a variable or passed to a function like any other value
- Ex: ['cat', 'bat', 'rat', 'elephant'],
[1,2,3,4,5]
- A list begins with an opening square bracket and ends with a closing square bracket, []

The List Data Type

➤ Values inside the list are also called items. Items are separated with commas (=they are comma-delimited).

```
>>> [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']
```

- The spam variable ❶ is still assigned only one value: the list value.
- But the list value itself contains other values.

Getting Individual Values in a List with Indexes

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

```
      ↗       ↗       ↖       ↖  
spam[0] spam[1] spam[2] spam[3]
```

- Consider a list: `spam= ['cat', 'bat', 'rat', 'elephant']`
- `spam[0]= 'cat', spam[1]= 'bat', spam[2]= 'rat', spam[3]='elephant'`
- **index:** The integer inside square brackets that follows the list
- 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.

Getting Individual Values in a List with Indexes

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



A diagram illustrating the mapping between list indices and their corresponding values. Four arrows point from the indices below to the elements in the list above: an arrow from `spam[0]` to `"cat"`, an arrow from `spam[1]` to `"bat"`, an arrow from `spam[2]` to `"rat"`, and an arrow from `spam[3]` to `"elephant"`.

➤ `spam[0]='cat', spam[1]='bat', spam[2]='rat',
spam[3]='elephant'`


➤ **Note:**

➤ First index is = 0,

➤ the last index is = size of the list - 1
= 4 - 1 = 3

Getting Individual Values in a List with Indexes

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



The diagram illustrates how to access individual elements in a list using indexes. Below the list definition, four indexes are shown: `spam[0]`, `spam[1]`, `spam[2]`, and `spam[3]`. 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".

➤ Note: The value `[]` is an empty list that contains no values, similar to `''` (empty string)

Getting Individual Values in a List with Indexes

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

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

```
'cat'
```

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

```
'bat'
```

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

```
'rat'
```

```
>>> 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.'
```


Getting Individual Values in a List with Indexes

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

```
>>> spam[10000]
```

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

```
File "<pyshell#9>", line 1, in <module>
```

```
spam[10000]
```

```
IndexError: list index out of range
```

➤ **IndexError error message** : occurs if you use an index that exceeds the number of values in your list value.

Getting Individual Values in a List with Indexes

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

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

```
'bat'
```

```
>>> spam[1.0]
```

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

```
File "<pyshell#13>", line 1, in <module>
```

```
spam[1.0]
```

```
TypeError: list indices must be integers or  
slices, not float
```

```
>>> spam[int(1.0)]
```

```
'bat'
```

➤ **TypeError error**
message : This error message will be thrown if index not = integer.
Indexes can be only integer values, not floats.

Getting Individual Values in a List with Indexes

➤ The values in these lists of lists can be accessed using multiple indexes

➤ Ex: `>>>spam[0][1]`
'bat'

➤ =the second value in the first list.

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

The **first index** dictates which **list value** to use, and the **second** indicates the **value within the list value**.

Getting Individual Values in a List with Indexes

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
```

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

```
['cat', 'bat']
```

```
>>> spam[0][1]
```

```
'bat'
```

```
>>> spam[1][4]
```

```
50
```

➤ Ex: >>>spam[0]

```
['cat', 'bat']
```

➤ =the full list value at
index=0

If you only use **one index**, the program will print the full list value at that index.

Getting a List from Another List USING Slices

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

```
>>> spam[0:4]
```

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

```
>>> spam[1:3]
```

```
['bat', 'rat']
```

```
>>> spam[0:-1]
```

```
['cat', 'bat', 'rat']
```

- Index gets a single value from a list
- Slice gets several values from a list, in the form of a new list.

➤ A slice evaluates to a new list value.

➤ A slice is typed between square brackets, like an index, but it has two integers separated by a colon.

Getting a List from Another List USING Slices

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

```
>>> spam[0:4]
```

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

```
>>> spam[1:3]
```

```
['bat', 'rat']
```

```
>>> spam[0:-1]
```

```
['cat', 'bat', 'rat']
```

- In a slice, the first integer is the index where 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.

➤ `spam[2]` is a list with an index (one integer).

➤ `spam[1:4]` is a list with a slice (two integers).

Getting a List from Another List USING Slices

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

- Leaving out the first index is = using 0, or the list starts from beginning of the list.
- Leaving out the second index will slice to the end of the list= list ends at the last item

➤ As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice.

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

- The len() function will return the number of items that are in a list value passed to it.
- The len() function counts the number of characters when string value is passed to it.

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

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

```
3
```


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

- Use an index of a list to change the value at that index.
- Ex: `spam[1] = 'aardvark'` means
“Assign the value at index 1 in the list `spam` to the string `'aardvark'`.”

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['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
- the * operator => replicates integer value times the given list when used with an integer.

```
>>> [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.
- The del statement can also be used on a simple variable to delete it.
- NameError error message : will be thrown if the variable is used after deleting it because the variable no longer exists.

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

Working with Lists

- a bad way to write code: to create many individual variables to store a group of similar values
- if the number of cats changes, this program will never be able to store more cats than the number of variables.
- These types of programs also have a lot of duplicate or nearly identical code in them

Ex: store the names of my cats:

```
catName1 = 'Zophie'
```

```
catName2 = 'Pooka'
```

```
catName3 = 'Simon'
```

```
catName4 = 'Lady Macbeth'
```

```
catName5 = 'Fat-tail'
```

```
catName6 = 'Miss Cleo'
```

Working with Lists

allmycats1.py:

```
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()
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)
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value.

Working with Lists

allMyCats2.py:

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) + ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name] # list concatenation
print('The cat names are:')
for name in catNames:
    print(' ' + name)
```

This new version uses a single list and can store any number of cats that the user types in.

Working with Lists

allMyCats2.py: Output:

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

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

The benefit of using a list is that data entered is in a structure, so such a program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

➤ a for loop repeats the code block once for each item in a list value.

the return value from `range(4)` is a sequence value $\sim [0, 1, 2, 3]$.

Ex:

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

The following program has the same output as the previous one:

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

Output:

0
1
2
3

Using for Loops with Lists

➤ A common Python technique is to use `range(len(someList))` with a for loop to iterate over the indexes of a list

➤ Ex:

```
>>> 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

Using for Loops with Lists

➤ Using `range(len(supplies))` in for loop is handy because the code in the loop can access the index `=i` and the value at that index `=as supplies[i]`.

➤ Ex:

```
>>> 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

`range(len(supplies))` will iterate through all the indexes of supplies, no matter how many items it contains.

The in and not in Operators

- **in** and **not in** operators: used to determine whether a value is or isn't in a list.
- Like other operators, **in** and **not in** are used in expressions and connect two values: a value to look for in a list and the list where it may be found.
- These expressions will evaluate to a Boolean value.

Ex:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
```

```
True
```

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

```
>>> 'cat' in spam
```

```
False
```

```
>>> 'howdy' not in spam
```

```
False
```

```
>>> 'cat' not in spam
```

```
True
```

The in and not in Operators

- Consider a program that lets the user type in a pet name
- and then checks to see whether the name is in a list of pets

Output:

Enter a pet name:

Footfoot

I do not have a pet named Footfoot

myPets.py:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
    if name not in myPets:
        print('I do not have a pet named ' +
name)
    else:
        print(name + ' is my pet.')
```

The Multiple Assignment Trick

- The multiple assignment trick (= tuple unpacking) is a shortcut that lets to assign multiple variables with the values in a list in one line of code.
- The number of variables and the length of the list must be exactly equal, or Python will give a **ValueError**:

```
>>> cat = ['fat', 'gray', 'loud']
```

```
>>> size = cat[0]
```

```
>>> color = cat[1]
```

```
>>> disposition = cat[2]
```

```
>>> cat = ['fat', 'gray', 'loud']
```

```
>>> size, color, disposition = cat
```

```
>>> 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

- 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.

Ex:

```
>>> 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

The `enumerate()` function is useful if both the item and the item's index in the loop's block are needed

random.choice() and random.shuffle()

- random module has a couple functions that accept lists for arguments.
- random.choice() function will return a randomly selected item from list.
- random.shuffle() function will reorder the items in a list.
- This function modifies the list in place, rather than returning a new list.

```
>>> import random
>>> pets = ['Dog', 'Cat', 'Moose']
>>> random.choice(pets)
'Dog'
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Cat'
```

```
>>> import random
>>> people = ['Alice', 'Bob', 'Carol', 'David']
>>> random.shuffle(people)
>>> people
['Carol', 'David', 'Alice', 'Bob']
>>> random.shuffle(people)
>>> people
['Alice', 'David', 'Bob', 'Carol']
```

Augmented Assignment Operators

➤ After assigning 42 to the variable spam, code used to increase the value in spam by 1 :

```
➤ >>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

Using Augmented Operator:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

- As a shortcut, use the augmented assignment operator += to do the same thing.
- There are augmented assignment operators for the +, -, *, /, and % Operators.

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>

Augmented Assignment Operators

- `+=` operator => used for string and list concatenation
- `*=` operator => used for string and list replication

```
>>> spam = 'Hello,'  
>>> spam += ' world!'  
>>> spam  
'Hello world!'  
>>> bacon = ['Zophie']  
>>> bacon *= 3  
>>> bacon  
['Zophie', 'Zophie', 'Zophie']
```

Methods

- A method is the same as a function, except it is “called on” a value.
- The method part comes after the value, separated by a period.
- Ex: if a list value were stored in spam, call the `index()` list method on that list :
 - `spam.index('hello').`
- Each data type has its own set of methods.
- Methods belong to a single data type.
- The list data type, has several useful methods for finding, adding, removing, and 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.

```
>>> 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
```

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

- When there are duplicates of the value in the list, the index of its first appearance is returned.

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
```

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

```
1
```

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

➤ To add new values to a list, use `append()` and `insert()` methods

➤ `append()` method call adds the argument to end of the list.

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

```
>>> spam.append('moose')
```

```
>>> spam
```

```
['cat', 'dog', 'bat', 'moose']
```

➤ The `insert()` method can insert a value at any index in the list.

➤ The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted.

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

```
>>> spam.insert(1, 'chicken')
```

```
>>> spam
```

```
['cat', 'chicken', 'dog', 'bat']
```

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

- code is `spam.append('moose')`
not `spam = spam.append('moose')`
`spam.insert(1, 'chicken')`,
not `spam = spam.insert(1, 'chicken')`
- `append()` and `insert()` don't return new value of `spam`. Rather, the list is modified in place.
- The return value of `append()` and `insert()` is `None`.
- `append()` and `insert()` methods are list methods and can be called only on list values, not on other values such as strings or integers.

```
>>> eggs = 'hello'
```

```
>>> eggs.append('world')
```

Traceback (most recent call last):

File "<pyshell#19>", line 1, in <module>

```
eggs.append('world')
```

AttributeError: 'str' object has no attribute 'append'

```
>>> bacon = 42
```

```
>>> bacon.insert(1, 'world')
```

Traceback (most recent call last):

File "<pyshell#22>", line 1, in <module>

```
bacon.insert(1, 'world')
```

AttributeError: 'int' object has no attribute 'insert'

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.
- Attempting to delete a value that does not exist in the list will result in a **ValueError** error.

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

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']  
>>> spam.remove('chicken')  
Traceback (most recent call last):  
File "<pyshell#11>", line 1, in <module>  
spam.remove('chicken')  
ValueError: list.remove(x): x not in list
```


Removing Values from Lists with the remove() Method

- If the value appears multiple times in the list, **only the first instance of the value** will be removed

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

- The del statement is good to use when you know the index of value you want to remove from the list.
- **The remove() method is useful when you know the value you want to remove from the list.**

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

- List of number values or list of strings can be sorted with the sort() method.
- Pass True for the reverse keyword argument to have sort() sort the values in reverse order.

```
>>> 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']
```

```
>>> spam.sort(reverse=True)
```

```
>>> spam
```

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

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

- Three important things about the sort() method:
 1. sort() method sorts the list in place; don't try to capture the return value by writing code like spam = spam.sort()
 2. Cannot sort lists that have both number values and string values in them, since Python doesn't know how to compare these values.

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
```

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

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

```
File "<pyshell#70>", line 1, in <module>
```

```
spam.sort()
```

```
TypeError: '<' not supported between  
instances of 'str' and 'int'
```

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

3. `sort()` uses “ASCIIbetical order” rather than actual alphabetical order for sorting strings.

- This means uppercase letters come before lowercase letters.
- Therefore, the lowercase a is sorted so that it comes after the uppercase Z.
- To sort the values in regular alphabetical order, pass `str.lower` for the `key` keyword argument in the `sort()` method call.

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers',  
'Carol', 'cats']  
>>> spam.sort()  
>>> spam  
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

```
>>> spam = ['a', 'z', 'A', 'Z']  
>>> spam.sort(key=str.lower)  
>>> spam  
['a', 'A', 'z', 'Z']
```

Reversing the Values in a List with reverse() Method

- to quickly reverse the order of the items in a list, call reverse() list method
- Like the sort() list method, reverse() doesn't return a list.
- This is why you write spam.reverse(), instead of spam = spam.reverse().

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

EXCEPTIONS TO INDENTATION RULES IN PYTHON

- the amount of indentation for a line of code tells Python what block it is in.
- There are some exceptions to this rule
- Ex: lists can actually span several lines in the source code file.
- Indentation of these lines does not matter; Python knows that list is not finished until it sees ending square bracket

```
spam = ['apples',  
        'oranges',  
        'bananas',  
        'cats']  
print(spam)
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

- split up a single instruction across multiple lines using the `\` line continuation character at the end.
- `\` = “This instruction continues on the next line.”
- The indentation on the line after a `\` line continuation is not significant
- These tricks are useful to rearrange long lines of Python code to be a bit more readable.

```
print('Four score and seven ' + \  
      'years ago...')
```

Sequence Data Types

➤ 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

➤ Many of the things you can do with lists can also be done with strings and other values of sequence types: **indexing**; **slicing**; using **for loops**, with **`len()`**, **`in`** and **`not in`** operators.

```
>>> name = 'Zophie'
```

```
>>> name[0]
```

```
'Z'
```

```
>>> name[-2]
```

```
'i'
```

```
>>> name[0:4]
```

```
'Zoph'
```

```
>>> 'Zo' in name
```

```
True
```

```
>>> 'z' in name
```

```
False
```

```
>>> 'p' not in name
```

```
False
```

```
>>> for i in name:
```

```
... print('* * * ' + i + ' * * *')
```

```
* * * Z * * *
```

```
* * * o * * *
```

```
* * * p * * *
```

```
* * * h * * *
```

```
* * * i * * *
```

```
* * * e * * *
```


Mutable and Immutable Data Types

- Lists and strings are different in an important way.
- A list value is a mutable data type: it can have values added, removed, or changed.
- However, a string is immutable: it cannot be changed.
- Trying to reassign a single character in a string results in a **TypeError** error,

```
>>> name = 'Zophie a cat'
```

```
>>> name[7] = 'the'
```

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

```
File "<pyshell#50>", line 1, in
```

```
<module>
```

```
name[7] = 'the'
```

```
TypeError: 'str' object does not  
support item assignment
```

Mutable and Immutable Data Types

- The proper way to “mutate” a string is to use **slicing and concatenation** to build a new string by copying from parts of the old string.

```
>>> name = 'Zophie a cat'
```

```
>>> newName = name[0:7] + 'the' + name[8:12]
```

```
>>> name
```

```
'Zophie a cat'
```

```
>>> newName
```

```
'Zophie the cat'
```

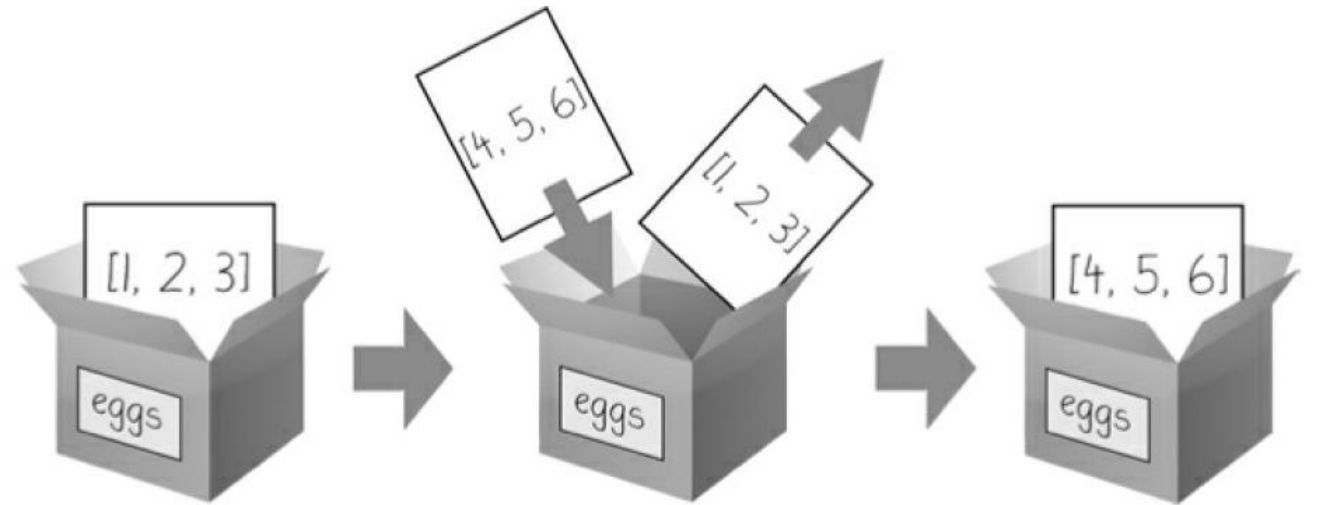
We used **[0:7]** and **[8:12]** to refer to the characters that we don't wish to replace.

Note that the original 'Zophie a cat' string is not modified, **because strings are immutable.**

Mutable and Immutable Data Types

➤ Although a list value is mutable, the second line in the following code does not modify the list `eggs`:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```



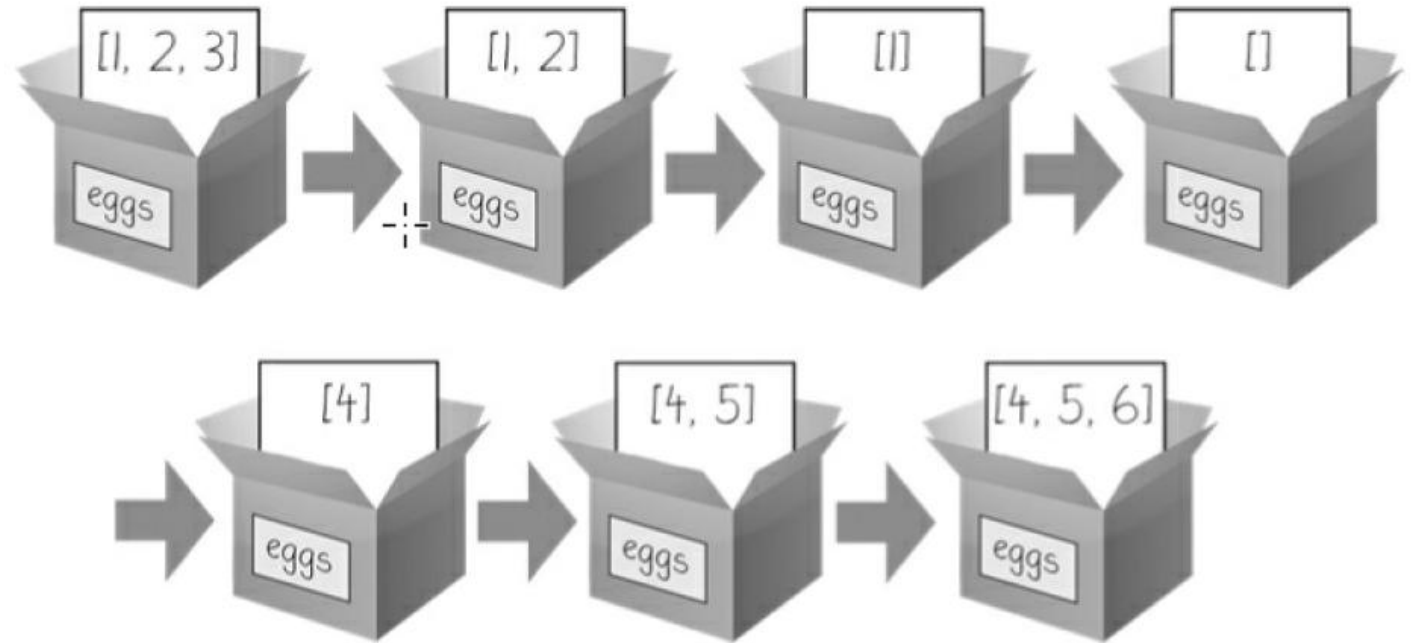
When `eggs = [4, 5, 6]` is executed, the contents of `eggs` are replaced with a new list value.

- The list value in `eggs` isn't being changed here; rather, an entirely new and different list value (`[4, 5, 6]`) is overwriting old list value (`[1, 2, 3]`).
- Changing a value of a mutable data type (Ex: `del` statement and `append()`) changes value in place, since variable's value is not replaced with a new list value.

Mutable and Immutable Data Types

- To actually modify the original list in eggs to contain [4, 5, 6], do something like this:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```



The `del` statement and the `append()` method modify the same list value in place.

The Tuple Data Type

tuple data type is almost identical to the list data type, except in two ways.

1. tuples are typed with parentheses, (), instead of square brackets, [].
2. tuples, like strings, are immutable, where as lists are mutable. **Tuples cannot have their values modified, appended, or removed.**

```
>>> eggs = ('hello', 42, 0.5)
```

```
>>> eggs[0]
```

```
'hello'
```

```
>>> eggs[1:3]
```

```
(42, 0.5)
```

```
>>> len(eggs)
```

```
3
```

```
>>> eggs = ('hello', 42, 0.5)
```

```
>>> eggs[1] = 99
```

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

```
File "<pyshell#5>", line 1, in <module>
```

```
eggs[1] = 99
```

```
TypeError: 'tuple' object does not  
support item assignment
```

The Tuple Data Type

- If only one value is present in a tuple, then indicate this by placing a trailing comma after the value inside the parentheses.
- Otherwise, Python will think you've just typed a value inside regular parentheses.
- The comma lets Python know this is a tuple value.
- Ex: type() function

```
>>> type(('hello',))
```

```
<class 'tuple'>
```

```
>>> type(('hello'))
```

```
<class 'str'>
```

The Tuple Data Type

- Tuples are immutable and their contents don't change
- Use tuples to convey reader that you don't intend for that sequence of values to change.
- Uses a tuple:
 1. If you need an ordered sequence of values that never changes.
 2. Since code using tuples run slightly faster than code using lists, can be used to implement some optimizations .

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

- `str(42)` will return `'42'`, the string representation of the integer 42
- Functions `list()` and `tuple()` will return list and tuple versions of the values passed to them.
- Note that the return value is of a different data type for the values passed:

Converting a tuple to a list is handy when a mutable version of a tuple value is needed

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


References

- Variables “store” strings and integer values.
- This is a simplification explanation of what Python is actually doing
- Technically, variables are storing references to the computer memory locations where the values are stored.
 - 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
 - When you copy the value in spam and assign it to the variable cheese, you are actually copying the reference.

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

References

- Both the spam and cheese variables refer to the 42 value in the computer's memory.
- 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.

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

References

- Lists don't work like variables, because lists are mutable, that is, list values can change.

Consider the code that will make this distinction easier to understand:

- The code touched only the cheese list, but both the cheese and spam lists have changed.

❶ `>>> spam = [0, 1, 2, 3, 4, 5]`

❷ `>>> cheese = spam` # The reference is being copied, not the list.

❸ `>>> cheese[1] = 'Hello!'` # This changes the list value.

`>>> spam`

`[0, 'Hello!', 2, 3, 4, 5]`

`>>> cheese` # The cheese variable refers to the same list.

`[0, 'Hello!', 2, 3, 4, 5]`

References

- When you create the list ❶, you assign a reference to it in the spam variable.
- But the next line ❷ copies only the list reference in spam to cheese, not the list value itself.
- This means the values stored in spam and cheese now both refer to the same list.
- There is only one underlying list because the list itself was never actually copied.

So when you modify the first element of cheese ❸, you are modifying the same list that spam refers to.

❶ >>> spam = [0, 1, 2, 3, 4, 5]

❷ >>> cheese = spam # The reference is being copied, not the list.

❸ >>> cheese[1] = 'Hello!' # This changes the list value.

>>> spam

[0, 'Hello!', 2, 3, 4, 5]

>>> cheese # The cheese variable refers to the same list.

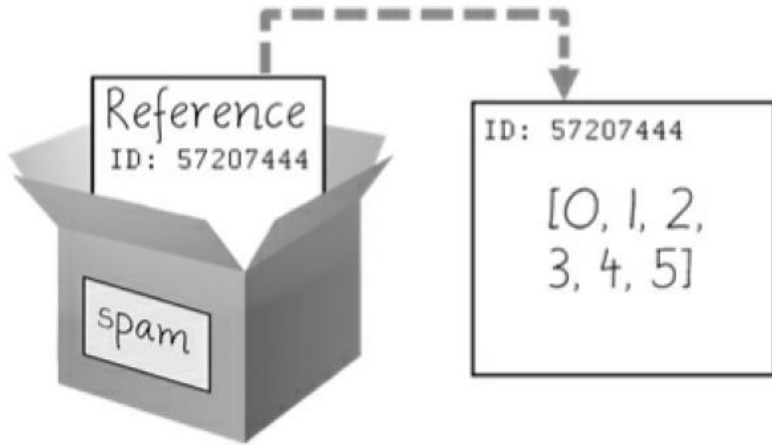
[0, 'Hello!', 2, 3, 4, 5]

References

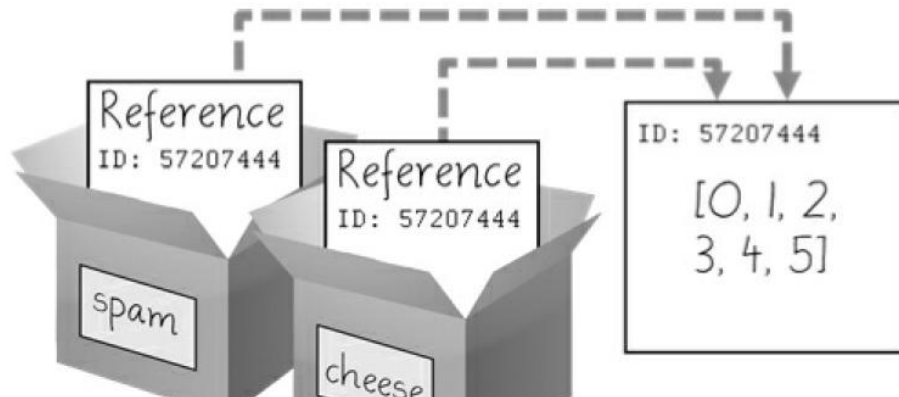
- Variables are like boxes that contain values.
- Lists in boxes aren't exactly accurate, because list variables don't actually contain lists—they contain references to lists.
- These references will have ID numbers that Python uses internally
- What happens when a list is assigned to the spam variable is visualized using boxes as a metaphor.

References

`spam = [0, 1, 2, 3, 4, 5]` stores a reference to a list, not the actual list.



`spam = cheese` copies the reference, not the list.



❶ `>>> spam = [0, 1, 2, 3, 4, 5]`

❷ `>>> cheese = spam` # The reference is being copied, not the list.

❸ `>>> cheese[1] = 'Hello!'` # This changes the list value.

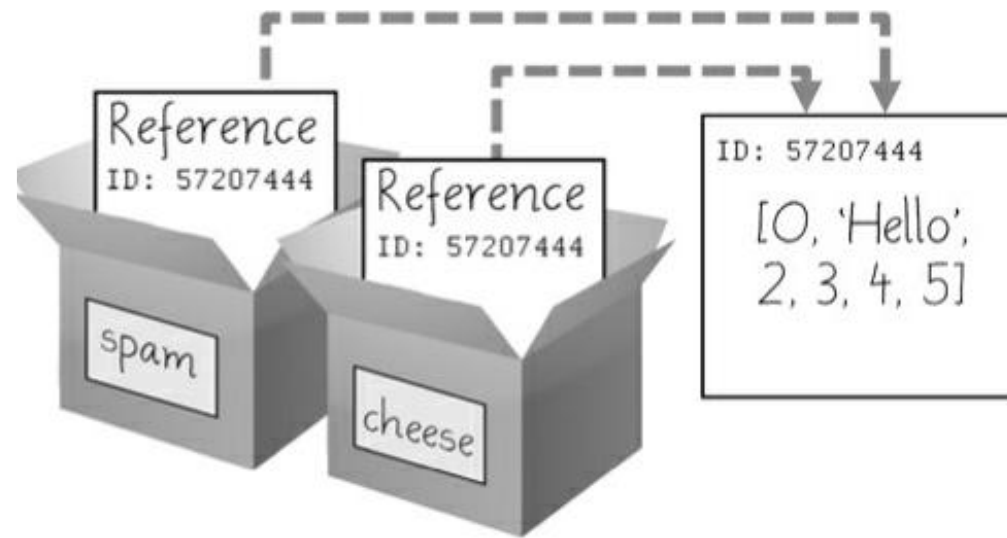
`>>> spam`

`[0, 'Hello!', 2, 3, 4, 5]`

`>>> cheese` # The cheese variable refers to the same list.

`[0, 'Hello!', 2, 3, 4, 5]`

`cheese[1] = 'Hello!'` modifies the list that both variables refer to.



Identity and the id() Function

- All values in Python have a unique identity that can be obtained with the `id()` function.
- `>>> id('Howdy') # returned number depend on machine.`
`44491136`
- When Python runs `id('Howdy')`, it creates the 'Howdy' string in the computer's memory.
- The numeric memory address where the string is stored is returned by the `id()` function.
- Python picks this address based on which memory bytes happen to be free on computer at the time, so it'll be different each time you run this code.

Identity and the id() Function

- Like all strings, 'Howdy' is immutable and cannot be changed.
- If you “change” the string in a variable, a new string object is being made at a different place in memory, and the variable refers to this new string.
- Ex:

```
>>> bacon = 'Hello'
>>> id(bacon)
44491136
>>> bacon += ' world!' # A new string is made from 'Hello' and ' world!'.
>>> id(bacon) # bacon now refers to a completely different string.
44609712
```


Identity and the id() Function

- Lists can be modified because they are mutable objects
- `append()` method doesn't create a new list object; it changes existing list object.
- This is known as “modifying the object in-place.”

```
>>> eggs = ['cat', 'dog'] # This creates a new list.
```

```
>>> id(eggs)
```

```
35152584
```

```
>>> eggs.append('moose') # append() modifies the list  
                        "in place".
```

```
>>> id(eggs) # eggs still refers to the same list as  
            before.
```

```
35152584
```

```
>>> eggs = ['bat', 'rat', 'cow'] # This creates a new list,  
                                which has a new identity.
```

```
>>> id(eggs) # eggs now refers to a completely  
            different list.
```

```
44409800
```

Identity and the id() Function

- If two variables refer to the same list (like spam and cheese in the previous section) and the list value itself changes, both variables are affected because they both refer to same list.
- The `append()`, `extend()`, `remove()`, `sort()`, `reverse()`, and other list methods modify their lists in place.
- Python's automatic garbage collector deletes any values not being referred to by any variables to free up memory.

Passing References

- References are particularly important for understanding how arguments get passed to functions
- When a function is called, the values of the arguments are copied to the parameter variables.
- For lists and dictionaries, this means a copy of the reference is used for the parameter.

passingReference.py:

```
def eggs(someParameter):  
    someParameter.append('Hello')  
    spam = [1, 2, 3]  
    eggs(spam)  
    print(spam)
```

Note that when `eggs()` is called, a return value is not used to assign a new value to `spam`.

Passing References

- Note that when `eggs()` is called, a return value is not used to assign a new value to `spam`.
- Instead, it modifies the list in place, directly.
- When run, this program produces the following output:

`[1, 2, 3, 'Hello']`

- Even though `spam` & `someParameter` contain separate references, they both refer to same list

`passingReference.py:`

```
def eggs(someParameter):  
    someParameter.append('Hello')  
spam = [1, 2, 3]  
eggs(spam)  
print(spam)
```

This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

copy Module's copy() and deepcopy() Functions

- Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value.
- For this, Python provides a module named copy that provides both the `copy()` and `deepcopy()` functions.
 - `copy.copy()`, can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference.

copy Module's copy() and deepcopy() Functions

copy.copy(),
can be used to
make a
duplicate copy
of a mutable
value like a list
or dictionary,
not just a copy
of a reference.

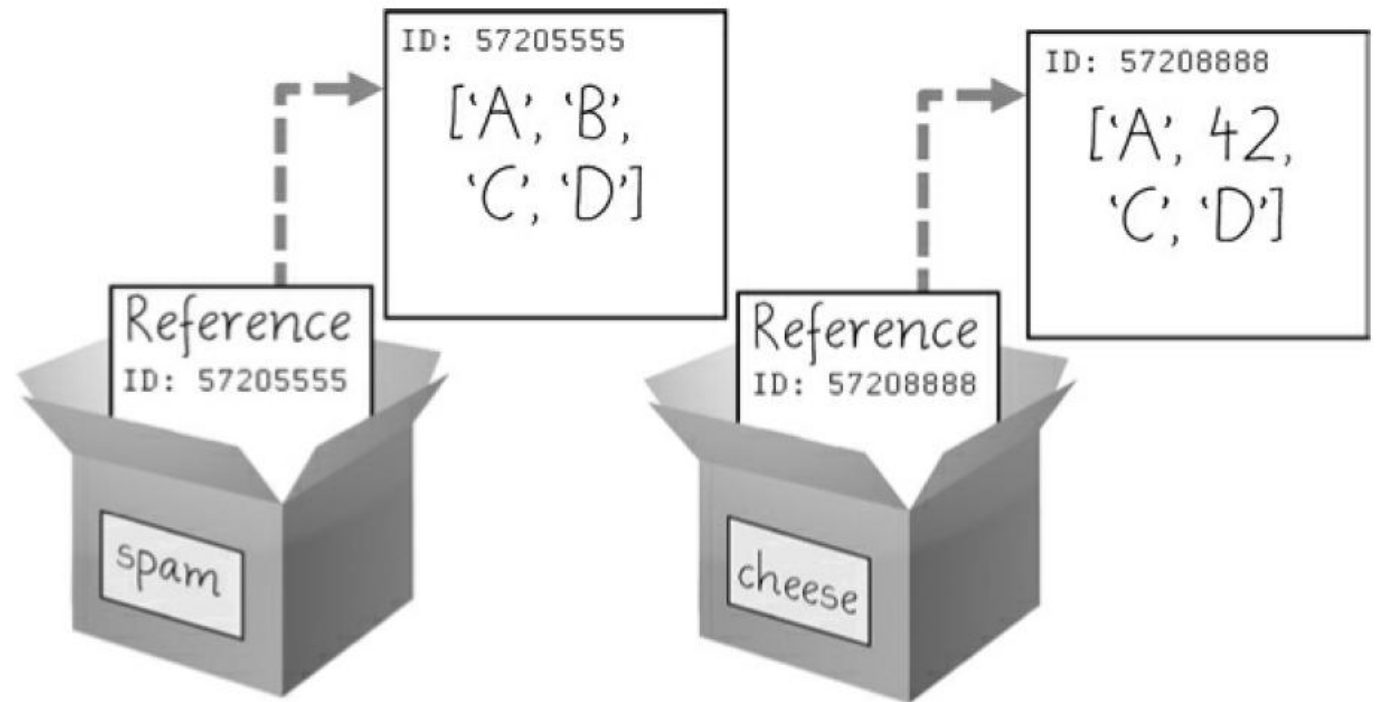
```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> id(spam)
44684232
>>> cheese = copy.copy(spam)
>>> id(cheese) # cheese is a different list with different
                identity.
44685832
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

copy Module's copy() and deepcopy() Functions

Now the spam and cheese variables refer to separate lists, which is why only the list in cheese is modified when you assign 42 at index 1.

reference ID numbers are no **longer** the same for both variables because the variables refer to independent lists.

cheese = copy.copy(spam) creates a second list that can be modified independently of first.



copy Module's copy() and deepcopy() Functions

- If the list you need to copy contains lists, then use the `copy.deepcopy()` function instead of `copy.copy()`.
- The `deepcopy()` function will copy these inner lists as well. A Short Program: Conway's Game

DICTIONARIES and STRUCTURING DATA

The Dictionary Data Type

- Dictionary data type, provides a flexible way to access and organize data
- Dictionary is a mutable collection of many values
- 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.
- dictionary is typed with braces, { }.

The Dictionary Data Type

```
>>> 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:

```
>>> myCat['size']
```

```
'fat'
```

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

```
'My cat has gray fur.'
```

The Dictionary Data Type

Ex:

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

- Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at 0 and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

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.

Order of items matters for determining whether two lists are the same.

- But it does not matter in what order the key-value pairs are typed in a dictionary.

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

False

- Because dictionaries are not ordered, they can't be sliced like

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

True

Dictionaries vs. Lists

- List's throws IndexError error message, when “out-of-range” is encountered
- Similarly, trying to access a key that does not exist in a dictionary will result in a KeyError error message.

Error message
shows up because
there is no 'color'
key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
```

```
>>> spam['color']
```

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

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

```
spam['color']
```

```
KeyError: 'color'
```

Dictionaries vs. Lists

- Though dictionaries are not ordered, the arbitrary values for the keys allows to organize data in powerful ways.
- Ex: Consider a program that stores data about friends' birthdays.
- Use a dictionary with the names as keys and the birthdays as values.
- Consider a program: birthdays.py

Dictionaries vs. Lists

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}
while True:
    print('Enter a name: (blank to quit)')
    name = input()
    if name == '':
        break
    ❷ if name in birthdays:
        ❸ print(birthdays[name] + ' is the birthday of ' +
name)
    else:
        print('I do not have birthday information for ' + name)
        print('What is their birthday?')
        bday = input()
        ❹ birthdays[name] = bday
        print('Birthday database updated.')
```

Enter a name: (blank to quit)

Alice

Apr 1 is the birthday of Alice

Enter a name: (blank to quit)

Eve

I do not have birthday
information for Eve

What is their birthday?

Dec 5

Birthday database updated.

Enter a name: (blank to quit)

Eve

Dec 5 is the birthday of Eve

Enter a name: (blank to quit)

ORDERED DICTIONARIES IN PYTHON 3.7

Dictionaries will remember the insertion order of their key-value pairs if you create a sequence value from them

Ex: note the order of items in the lists made from the eggs and ham dictionaries matches the order in which they were entered:

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

```
>>> list(eggs)
```

```
['name', 'species', 'age']
```

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

```
>>> list(ham)
```

```
['species', 'age', 'name']
```

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

- **keys(), values(), and items():** are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values.
- 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.
- Here, a for loop iterates over each of the values in the spam dictionary.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
...     print(v)
red
42
```

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

- A for loop can also iterate over the keys or both keys and values.
- 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.
- Note that the values in the dict_items value returned by the items() method are tuples of the key and value.

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

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

- If you want a true list from one of these methods, pass its list-like return value to the list() function.
- The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age'].

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

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

- Use the multiple assignment trick in a for loop to assign the key and value to separate variables.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
...     print('Key: ' + k + ' Value: ' + str(v))
Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

➤ `in` and `not in` operators are used to check whether a value exists in a list

➤ Similarly use these operators to see whether a certain 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
```

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

```
True
```

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

```
False
```

Checking Whether a Key or Value Exists in a Dictionary

➤ `'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.

```
>>> spam = {'name': 'Zophie', 'age': 7}
```

```
>>> 'name' in spam.keys()
```

```
True
```

```
>>> 'Zophie' in spam.values()
```

```
True
```

```
>>> 'color' in spam.keys()
```

```
False
```

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

```
True
```

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

```
False
```

The get() Method

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

```
>>> 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.'
```


The get() Method

- 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,

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

```
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
```

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

```
'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
```

```
KeyError: 'eggs'
```

The.setdefault() Method

- `setdefault()` method is used to set a value in a dictionary for a certain key only if that key does not already have a value.
- in `and` `not in` are used as follows: ????????check and put output here
 `spam = {'name': 'Pooka', 'age': 5}`
 `if 'color' not in spam:`
 `spam['color'] = 'black'`

The.setdefault() Method

- `setdefault()` method is used to set a value in a dictionary for a certain key only if that key does not already have a value.
- `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 second argument is 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.

```
spam = {'name': 'Pooka', 'age': 5}
```

```
if 'color' not in spam:
```

```
    spam['color'] = 'black'
```

The.setdefault() Method

- `setdefault()` method is used to set a value in a dictionary for a certain key **only** if that key does not already have a value.
- The first argument passed to the method is the key to check for, and second argument is value to set at that key if the key does not exist.
- If the key does exist, `setdefault()` method returns the key's value.

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The.setdefault() Method

➤ The first time `setdefault()` is called, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`.

➤ The method returns value `'black'` because this is now the value set for the key `'color'`.

➤ When `spam.setdefault('color', 'white')` is called next, the value for that key is not changed to `'white'`, because `spam` already has a key named `'color'`.

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The.setdefault() Method

characterCount.py:

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

```
count = {}
```

```
for character in message:
```

```
    ❶ count.setdefault(character, 0)
```

```
    ❷ count[character] = count[character] + 1
```

```
print(count)
```

The program loops over each character in the message variable's string, counting **how often each character appears**.

The `setdefault()` method call ❶ ensures that the key is in the count dictionary (with a default value of 0) so the program doesn't throw a **KeyError error** when `count[character] = count[character] + 1` is executed ❷.

Pretty Printing

characterCount.py:

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

```
count = {}
```

```
for character in message:
```

```
    ❶ count.setdefault(character, 0)
```

```
    ❷ count[character] = count[character] + 1
```

```
print(count)
```

Output:

```
{ ' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c':  
3, 'b': 1, 'e': 5, 'd': 3, 'g': 2,  
'i': 6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p':  
1, 's': 3, 'r': 5, 't': 6,  
'w': 2, 'y': 1}
```

- In Output, lowercase letter c appears 3 times, space character appears 13 times & uppercase letter A appears 1 time.

- This program will work no matter what string is inside the message variable, even if the string is millions of characters long.

Pretty Printing

- import `pprint` module into programs, to have access to `pprint()` and `pformat()` functions that will “pretty print” a dictionary’s values.
- Helpful when a cleaner display of the items is needed in a dictionary than what `print()` provides.

```
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)
```

- Consider `prettyCharacterCount.py`.

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

This time, when the program is run, the output looks much cleaner, with the keys sorted.

Nested Dictionaries and Lists

- As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists.
- Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.
- Ex: Consider a program that uses a dictionary that contains other dictionaries of 'what items guests are bringing to a picnic'.
- The `totalBrought()` function can read this data structure and calculate the total number of an item being brought by all the guests.

Nested Dictionaries and Lists

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
    ❶ for k, v in guests.items():
    ❷ numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies ' + str(totalBrought(allGuests, 'apple pies')))
```

Nested Dictionaries and Lists

- Inside the `totalBrought()` function, the for loop iterates over the key-value pairs in `guests` ❶.
- Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`.
- If the `item` parameter exists as a key in this dictionary, its value (the quantity) is added to `numBrought` ❷.
- If it does not exist as a key, the `get()` method returns 0 to be added to `numBrought`.

Nested Dictionaries and Lists

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}
def totalBrought(guests, item):
    numBrought = 0
    ❶ for k, v in guests.items():
    ❷ numBrought = numBrought + v.get(item, 0)
    return numBrought
print('Number of things being brought:')
print(' - Apples ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies ' + str(totalBrought(allGuests, 'apple pies')))
```

Output:

Number of things being brought

- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies 1

Nested Dictionaries and Lists

- This may seem like such a simple thing to model that you may feel no need to bother with writing a program to do it.
- But realize that this same `totalBrought()` function could easily handle a dictionary that contains `thousands of guests`, each bringing `thousands of different picnic items`.
- Then having this information in a data structure along with the `totalBrought()` function would save you a lot of time!
- Model things with data structures in whatever way needed, as long as the rest of the code in your program can work with the data model correctly.
- When you first begin programming, don't worry so much about the "right" way to model data.
- As you gain more experience, you may come up with more efficient models.
- Important thing is that the data model works for your program's needs.

Summary

- Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries.
- Dictionaries are useful because you can map one item (key) to another (value), as opposed to lists, which simply contain a series of values in order
- Values inside a dictionary are accessed using square brackets just as with lists.
- Instead of an integer index, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples.
- By organizing a program's values into data structures, you can create representations of real-world objects.