Sequences: Lists and Tuples

1. Introduction

2. List

3. Tuple

4. Reference

One more topic you'll need to understand is the `list` data type and its cousin, the `tuple`. `Lists` and `tuples` can contain multiple values, which makes writing programs that handle large amounts of data easier.

One more topic you'll need to understand is the `list` data type and its cousin, the `tuple`. `Lists` and `tuples` can contain multiple values, which makes writing programs that handle large amounts of data easier.

These data types are called ***containers***, meaning they are objects that "contain" other objects. They each have some important distinguishing properties and come with their own set of function called ***methods*** for interacting with others.

One more topic you'll need to understand is the `list` data type and its cousin, the `tuple`. `Lists` and `tuples` can contain multiple values, which makes writing programs that handle large amounts of data easier.

These data types are called ***containers***, meaning they are objects that "contain" other objects. They each have some important distinguishing properties and come with their own set of function called ***methods*** for interacting with others.

`List` and `tuple` belong to ***sequence*** data types, which means they represent **ordered collections of items**. They share the same characteristic as `string` and the `range` object returned by `range()` function.

List

In a `string`, the values are characters; in a `list`, they can be any type. The values in a `list` are called ***elements*** or sometimes ***items***. Items are separated with commas.

In a `string`, the values are characters; in a `list`, they can be any type. The values in a `list` are called ***elements*** or sometimes ***items***. Items are separated with commas.

source: https://favtutor.com/blogs/list-vs-dictionary

There are several ways to create a new `list`; the simplest is to enclose the elements in square brackets (“[” and ”]”). A `list` that contains no elements is called an empty `list`; you can create one with empty brackets.

There are several ways to create a new `list`; the simplest is to enclose the elements in square brackets ("[" and "]"). A `list` that contains no elements is called an empty `list`; you can create one with empty brackets.

In [2]:
```
type([])
```

Out[2]:
```
list
```

There are several ways to create a new `list`; the simplest is to enclose the elements in square brackets (“[” and ”]”). A `list` that contains no elements is called an empty `list`; you can create one with empty brackets.

In [2]:
```
type([])
```

Out[2]:
```
list
```

In [3]:
```
type([10, 20, 30, 40]), type(['calculus', 'introduction to mathematics', 'com
```

Out[3]:
```
(list, list)
```

There are several ways to create a new `list`; the simplest is to enclose the elements in square brackets ("[" and "]"). A `list` that contains no elements is called an empty `list`; you can create one with empty brackets.

In [2]:
```
type([])
```

Out[2]:
```
list
```

In [3]:
```
type([10, 20, 30, 40]), type(['calculus', 'introduction to mathematics', 'com
```

Out[3]:
```
(list, list)
```

The first example is a list of four integers and the second is a list of four strings.

Getting Individual Values in a List with Indexes

You can reference a `list` item by writing the `list`'s name followed by the element's ***index*** (that is, its position number) enclosed in square brackets (`[]`, known as the ***subscript operator*** or ***bracket operator***). Remember that the <u>indices start at 0</u>:

You can reference a `list` item by writing the `list`'s name followed by the element's ***index*** (that is, its position number) enclosed in square brackets (`[]`, known as the ***subscript operator*** or ***bracket operator***). Remember that the indices start at 0:

In [4]:

```
subjects = ['calculus', 'introduction to mathematics', 'computer programming'
print(subjects[0])
print(subjects[3])
```

```
calculus
linear algebra
```

You can reference a `list` item by writing the `list`'s name followed by the element's ***index*** (that is, its position number) enclosed in square brackets (`[]`, known as the ***subscript operator*** or ***bracket operator***). Remember that the indices start at 0:

In [4]:
```
subjects = ['calculus', 'introduction to mathematics', 'computer programming'
print(subjects[0])
print(subjects[3])
```

```
calculus
linear algebra
```

Note that the first index is 0, the last index is one less than the size of the `list`; a `list` of four items has 3 as its last index.

Python will give you an `IndexError` error message (which is a type of runtime error) if you use an index that exceeds the number of values in your list value.

Python will give you an `IndexError` error message (which is a type of runtime error) if you use an index that exceeds the number of values in your list value.

In [5]:
```
print(subjects[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_5960\2337914966.py in <module>
----> 1 print(subjects[4])

IndexError: list index out of range
```

Python will give you an `IndexError` error message (which is a type of runtime error) if you use an index that exceeds the number of values in your list value.

In [5]:
```
print(subjects[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_5960\2337914966.py in <module>
----> 1 print(subjects[4])

IndexError: list index out of range
```

The elements of a `list` don't have to be the same type. The following `list` contains a `string`, a `float`, an `integer`, and another `list`:

Python will give you an `IndexError` error message (which is a type of runtime error) if you use an index that exceeds the number of values in your list value.

In [5]:
```
print(subjects[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5960\2337914966.py in <module>
----> 1 print(subjects[4])

IndexError: list index out of range
```

The elements of a `list` don't have to be the same type. The following `list` contains a `string`, a `float`, an `integer`, and another `list`:

In [6]:
```
spam = ['spam', 2.0, 5, [10, 20]]
```

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

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

In [7]:
```
spam[3][1] # spam = ['spam', 2.0, 5, [10, 20]]
```

Out[7]:
```
20
```

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

In [7]:
```
spam[3][1] # spam = ['spam', 2.0, 5, [10, 20]]
```

Out[7]:
```
20
```

The first index dictates which items in the outer `list` to use, and the second indicates the value within the inner `list`. If you only use one index like `spam[3]`, the program will print the entire list value at that index.

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

In [7]:
```
spam[3][1] # spam = ['spam', 2.0, 5, [10, 20]]
```

Out[7]:
```
20
```

The first index dictates which items in the outer `list` to use, and the second indicates the value within the inner `list`. If you only use one index like `spam[3]`, the program will print the entire list value at that index.

In [8]:
```
spam[3]
```

Out[8]:
```
[10, 20]
```

In [9]:

```python
display_quiz(path+"list1.json", max_width=800)
```

What is printed by the following statements?

```python
alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]
print(alist[5])
```

dog	False
3.14	[ ]

In [10]:

```
display_quiz(path+"list2.json", max_width=800)
```

Which of the following correctly uses indexing? Assume that a is a list or string.

a[]	t = a[0]
x = [8]	w = [a]

Negative Indexes and the `len()` function

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.

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.

In [11]:
```
print(subjects[-1]) # subjects = ['calculus', 'introduction to mathematics',
print(subjects[-2])
```

```
linear algebra
computer programming
```

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.

In [11]:
```
print(subjects[-1]) # subjects = ['calculus', 'introduction to mathematics',
print(subjects[-2])
```

```
linear algebra
computer programming
```

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

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.

In [11]:
```
print(subjects[-1]) # subjects = ['calculus', 'introduction to mathematics',
print(subjects[-2])
```

```
linear algebra
computer programming
```

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

In [12]:
```
len(subjects)
```

Out[12]:
```
4
```

Getting a sublist 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` as a **new list**. A slice is typed between square brackets, like an index, but has two integers separated by a colon.

Just as an index can get a single value from a `list`, a ***slice*** can get several values from a `list` as a **new list**. A slice is typed between square brackets, like an index, but has two integers separated by a colon.

- `subjects[2]` is a list with an index.
- `subjects[1:3]` is a list with a slice.

Just as an index can get a single value from a `list`, a ***slice*** can get several values from a `list` as a **new list**. A slice is typed between square brackets, like an index, but has two integers separated by a colon.

- `subjects[2]` is a list with an index.
- `subjects[1:3]` is a list with a slice.

The slice operator `[n:m]` returns the part of the string starting with the element at index `n` and go up to but not including the element at index `m`. A slice evaluates to a new `list`!

In [13]:

```
subjects = ['calculus', 'introduction to mathematics', 'computer programming'
print(subjects[0:3])
print(subjects[1:-1])
```

```
['calculus', 'introduction to mathematics', 'computer programming']
['introduction to mathematics', 'computer programming']
```

In [13]:

```
subjects = ['calculus', 'introduction to mathematics', 'computer programming'
print(subjects[0:3])
print(subjects[1:-1])
```

```
['calculus', 'introduction to mathematics', 'computer programming']
['introduction to mathematics', 'computer programming']
```

As a shortcut, you can leave out one or both indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0 or the beginning of the `list`. Leaving out the second index is the same as using the length of the `list`, which will slice to the end of the `list`.

In [13]:
```
subjects = ['calculus', 'introduction to mathematics', 'computer programming'
print(subjects[0:3])
print(subjects[1:-1])
```

```
['calculus', 'introduction to mathematics', 'computer programming']
['introduction to mathematics', 'computer programming']
```

As a shortcut, you can leave out one or both indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0 or the beginning of the `list`. Leaving out the second index is the same as using the length of the `list`, which will slice to the end of the `list`.

In [14]:
```
# subjects = ['calculus', 'introduction to mathematics', 'computer programmin
print(subjects[:3]) # same as subjects[0:3]
print(subjects[1:]) # same as subjects[1:len(s)]
print(subjects[:]) # same as s[0:len(s)]
```

```
['calculus', 'introduction to mathematics', 'computer programming']
['introduction to mathematics', 'computer programming', 'linear algebr
a']
['calculus', 'introduction to mathematics', 'computer programming', 'l
inear algebra']
```

Just like `range()`, slicing has the optional third index that can be used to specify the step.

Just like `range()`, slicing has the optional third index that can be used to specify the step.

In [15]:

```
# subjects = ['calculus', 'introduction to mathematics', 'computer programmin
print(subjects[::2]) # Note the default step is 1
print(subjects[::-1]) # Reverse the order of the list, subjects[len(subjects)
```

```
['calculus', 'computer programming']
['linear algebra', 'computer programming', 'introduction to mathematic
s', 'calculus']
```

```
In [16]: display_quiz(path+"slice.json", max_width=800)
```

What is printed by the following statements?

```
alist = [3, 67, "cat", [56, 57, "dog"], 2.5, 3.14, False]
print(alist[4:])
```

- [[56, 57, "dog"], 2.5, 3.14, False]
- [2.5, 3.14, False]
- [2.5, 3.14]
- 2.5

Changing Values in a `List` with Indexes

Unlike `strings`, `lists` are ***mutable*** because you can reassign an item in a `list`. When the bracket operator appears on the left side of an assignment, it identifies the element of the `list` that will be assigned. An assignment to an element of a list is called ***item assignment***:

Unlike `strings`, `lists` are ***mutable*** because you can reassign an item in a `list`. When the bracket operator appears on the left side of an assignment, it identifies the element of the `list` that will be assigned. An assignment to an element of a list is called ***item assignment***:

In [17]:
```
numbers = [17, 123, 42, 7]
numbers[1] = 5
print(numbers)
```

```
[17, 5, 42, 7]
```

Unlike `strings`, `lists` are ***mutable*** because you can reassign an item in a `list`. When the bracket operator appears on the left side of an assignment, it identifies the element of the `list` that will be assigned. An assignment to an element of a list is called ***item assignment***:

In [17]:
```
numbers = [17, 123, 42, 7]
numbers[1] = 5
print(numbers)
```

```
[17, 5, 42, 7]
```

The first element of numbers, which used to be 123, is now 5.

`List` Concatenation and `List` Replication

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

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

In [18]:
```
[1, 2, 3] + ['A', 'B', 'C']
```

Out[18]:
```
[1, 2, 3, 'A', 'B', 'C']
```

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

In [18]:
```
[1, 2, 3] + ['A', 'B', 'C']
```

Out[18]:
```
[1, 2, 3, 'A', 'B', 'C']
```

In [19]:
```
['X', 'Y', 'Z'] * 3
```

Out[19]:
```
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
```

In [20]:

```python
display_quiz(path+"concate.json", max_width=800)
```

What is printed by the following statements?

```python
alist = [1,3,5]
blist = [2,4,6]
print(alist + blist)
```

6	[1,3,5,2,4,6]
[1,2,3,4,5,6]	[3,7,11]

Removing Values from Lists with `del` Statements

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

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

In [21]:
```
t = ['a', 'b', 'c', 'd', 'e']
del t[1] # using index
print(t)
```

```
['a', 'c', 'd', 'e']
```

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

In [21]:
```
t = ['a', 'b', 'c', 'd', 'e']
del t[1] # using index
print(t)
```

```
['a', 'c', 'd', 'e']
```

We can also delete multiple adjacent elements using slicing:

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

In [21]:
```
t = ['a', 'b', 'c', 'd', 'e']
del t[1] # using index
print(t)
```

```
['a', 'c', 'd', 'e']
```

We can also delete multiple adjacent elements using slicing:

In [22]:
```
del t[1:3]
print(t)
```

```
['a', 'e']
```

`List` traversal

In Chapter 2, you have learned about using `for` loops to execute a block of code a certain number of times. **Technically, a `for` loop repeats the code block once for each item in a sequence.** We will refer to this type of sequence iteration as ***iteration by item***.

In Chapter 2, you have learned about using `for` loops to execute a block of code a certain number of times. **Technically, a `for` loop repeats the code block once for each item in a sequence.** We will refer to this type of sequence iteration as ***iteration by item***.

In [23]:
```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

In Chapter 2, you have learned about using `for` loops to execute a block of code a certain number of times. **Technically, a `for` loop repeats the code block once for each item in a sequence.** We will refer to this type of sequence iteration as ***iteration by item***.

In [23]:
```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

In [24]:
```
print(range(4))
list(range(4))
```

```
range(0, 4)
```

Out[24]:
```
[0, 1, 2, 3]
```

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

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

In [25]:
```
for i in [0, 1, 2, 3]:
    print(i)
```

```
0
1
2
3
```

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

In [25]:
```
for i in [0, 1, 2, 3]:
    print(i)
```

```
0
1
2
3
```

In [26]:
```
for subject in subjects: # subjects = ['calculus', 'introduction to mathemati
    print(subject)
```

```
calculus
introduction to mathematics
computer programming
linear algebra
```

This works well if you only need to read the elements of the `list`. But you need the indices if you want to write or update the elements. A common way to do that is to combine the functions `range()` and `len()`:

This works well if you only need to read the elements of the `list`. But you need the indices if you want to write or update the elements. A common way to do that is to combine the functions `range()` and `len()`:

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

This works well if you only need to read the elements of the `list`. But you need the indices if you want to write or update the elements. A common way to do that is to combine the functions `range()` and `len()`:

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

In [27]:

```
numbers = [17, 5, 42, 7]
for i in range(len(numbers)):
    print(i, numbers[i])
    numbers[i] = numbers[i]**2

print(numbers)
```

```
0 17
1 5
2 42
3 7
[289, 25, 1764, 49]
```

The `in` and `not in` Operators

You can determine whether an object is or isn’t in a `list` with the `in` and `not in` operators. These expressions will evaluate to a `Boolean` value.

You can determine whether an object is or isn't in a `list` with the `in` and `not in` operators. These expressions will evaluate to a `Boolean` value.

In [28]:
```
print('howdy' in ['hello', 'hi', 'howdy', 'heyas'])
print('English' not in subjects)
```

```
True
True
```

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 and the item itself.**

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 and the item itself.**

In [29]:

```
numbers = [17, 5, 42, 7]
print(list(enumerate(numbers)))

for i, number in enumerate(numbers):
    print(i, number)
    numbers[i] = number**2

print(numbers)
```

```
[(0, 17), (1, 5), (2, 42), (3, 7)]
0 17
1 5
2 42
3 7
[289, 25, 1764, 49]
```

Methods of the `list`

A ***method***, introduced in Chapter 1, is the same as a function, except it is "called on" an object. For example, if a `list` object 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 object, separated by a period.

A ***method***, introduced in Chapter 1, is the same as a function, except it is "called on" an object. For example, if a `list` object 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 object, separated by a period.

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`.

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

`append()` adds a new element to the end of a `list`:

`append()` adds a new element to the end of a `list`:

In [32]:
```
t = ['a', 'b', 'c']
t.append('d') # not t = t.append('d')
t # in-place operation!
```

Out[32]:
```
['a', 'b', 'c', 'd']
```

`append()` adds a new element to the end of a `list`:

In [32]:
```
t = ['a', 'b', 'c']
t.append('d') # not t = t.append('d')
t # in-place operation!
```

Out[32]:
```
['a', 'b', 'c', 'd']
```

The previous `append()` method call adds the argument to the end of the `list`. The `insert()` method can insert an element 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.

`append()` adds a new element to the end of a `list`:

In [32]:
```
t = ['a', 'b', 'c']
t.append('d') # not t = t.append('d')
t # in-place operation!
```

Out[32]:
```
['a', 'b', 'c', 'd']
```

The previous `append()` method call adds the argument to the end of the `list`. The `insert()` method can insert an element 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.

In [33]:
```
t = ['a', 'b', 'c']
t.insert(1,'e') # not t = t.insert(1, 'e')
t # in-place operation!
```

Out[33]:
```
['a', 'e', 'b', 'c']
```

Methods belong to a single data type. The `append()` and `insert()` methods are `list` methods and can be called only on `list` object, not on other objects such as `strings` or `integers`.

Methods belong to a single data type. The `append()` and `insert()` methods are `list` methods and can be called only on `list` object, not on other objects such as `strings` or `integers`.

In [34]:
```
eggs = 'hello'
eggs.append('world')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5960\3759910952.py in <module>
      1 eggs = 'hello'
----> 2 eggs.append('world')

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

Adding all the elements of a `List` to the end of `List` with the `extend()` Methods

Use `list` method `extend()` to add **all the elements of another sequence** to the end of a list:

Use `list` method `extend()` to add **all the elements of another sequence** to the end of a list:

In [35]:
```
color_names = ['orange', 'yellow', 'green']
color_names.extend(['indigo', 'violet']) # equivalent to color_names += ['ind
```

Use `list` method `extend()` to add **all the elements of another sequence** to the end of a list:

```
In [35]: color_names = ['orange', 'yellow', 'green']
         color_names.extend(['indigo', 'violet']) # equivalent to color_names += ['ind
```

```
In [36]: color_names
```

```
Out[36]: ['orange', 'yellow', 'green', 'indigo', 'violet']
```

In [37]:

```python
display_quiz(path+"append.json", max_width=800)
```

What is printed by the following statements?

```python
my_list = [1, 2, 3]
my_list.append([4, 5])
my_list.extend([6, 7])
print(my_list)
```

- [1, 2, 3, [4, 5], 6, 7]
- [1, 2, 3, [4, 5, 6, 7]]
- [1, 2, 3, 4, 5, 6, 7]
- [1, 2, 3, 4, 5, [6, 7]]

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

The `remove()` method will pass the object to be removed from the `list` when it is called:

The `remove()` method will pass the object to be removed from the `list` when it is called:

In [38]:

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

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

```
In [39]: display_quiz(path+"remove.json", max_width=800)
```

What is printed by the following statements?

```
my_list = [10, 20, 30, 40, 20]
del my_list[1]
my_list.remove(20)
print(my_list)
```

[10, 40, 20]	[10, 20, 30, 40]
[10, 30, 40, 20]	[10, 30, 40]

Sorting the elements in a `List` with the `sort()` Method

Lists of numbers or lists of strings can be sorted with the `sort()` method:

Lists of numbers or lists of strings can be sorted with the `sort()` method:

In [40]:
```
spam = [2, 5, 3.14, 1, -7]
spam.sort() # The default behavior is sorting in ascending order
print(spam)

spam = ['ants', 'cats', 'dogs', 'badgers', 'Elephants']
spam.sort()
print(spam)
```

```
[-7, 1, 2, 3.14, 5]
['Elephants', 'ants', 'badgers', 'cats', 'dogs']
```

Lists of numbers or lists of strings can be sorted with the `sort()` method:

In [40]:

```
spam = [2, 5, 3.14, 1, -7]
spam.sort() # The default behavior is sorting in ascending order
print(spam)

spam = ['ants', 'cats', 'dogs', 'badgers', 'Elephants']
spam.sort()
print(spam)
```

```
[-7, 1, 2, 3.14, 5]
['Elephants', 'ants', 'badgers', 'cats', 'dogs']
```

Note that `sort()` uses “ASCII order” rather than 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.

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order.

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order.

In [41]:
```
spam.sort(reverse=True) # Sort in descending order
print(spam)
```

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

Searching an element in a `List` with the `index()` Method

`List` objects have an `index()` method that accepts an argument, and if that argument exists in the list, the index of the argument is returned. If the argument isn’t in the `list`, then `Python` produces a `ValueError` error.

`List` objects have an `index()` method that accepts an argument, and if that argument exists in the list, the index of the argument is returned. If the argument isn't in the `list`, then `Python` produces a `ValueError` error.

In [42]:
```
spam = ['hello', 'hi', 'howdy', 'heyas']
spam.index('hi')
```

Out[42]:
```
1
```

`List` objects have an `index()` method that accepts an argument, and if that argument exists in the list, the index of the argument is returned. If the argument isn't in the `list`, then `Python` produces a `ValueError` error.

In [42]:

```
spam = ['hello', 'hi', 'howdy', 'heyas']
spam.index('hi')
```

Out[42]:

```
1
```

In [43]:

```
spam = ['hello', 'hi', 'howdy', 'heyas']
spam.index('world')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5960\4200790458.py in <module>
      1 spam = ['hello', 'hi', 'howdy', 'heyas']
----> 2 spam.index('world')

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

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

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

```
In [44]: spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
         spam.index('Pooka')

Out[44]: 1
```

Numerical functions for `list`

There are a number of built-in functions that can be used on `lists` that allow you to quickly look through a `list` without writing your own loops:

There are a number of built-in functions that can be used on `lists` that allow you to quickly look through a `list` without writing your own loops:

In [45]:
```
nums = [3, 41, 12, 9, 74, 15]
print(len(nums))
```

```
6
```

There are a number of built-in functions that can be used on `lists` that allow you to quickly look through a `list` without writing your own loops:

In [45]:
```
nums = [3, 41, 12, 9, 74, 15]
print(len(nums))
```

```
6
```

In [46]:
```
print(max(nums))
print(min(nums))
print(sum(nums))
```

```
74
3
154
```

`List` Comprehensions

Consider how you might make a `list` of the first 10 square numbers (that is, the square of each integer from 1 through 10).

Consider how you might make a `list` of the first 10 square numbers (that is, the square of each integer from 1 through 10).

In [47]:

```
squares = []
for value in range(1,11):
    squares.append(value**2)
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Consider how you might make a `list` of the first 10 square numbers (that is, the square of each integer from 1 through 10).

In [47]:

```
squares = []
for value in range(1,11):
    squares.append(value**2)
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

But a ***list comprehension*** allows you to generate this same list in just one line of code. A list comprehension combines the `for` loop and the creation of new elements into one line, and automatically appends each new element!

In [48]:

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [48]:
```python
squares = [value**2 for value in range(1, 11)]
print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To use this syntax

- Begin with a descriptive name for the `list`, such as `squares`.
- Next, open a set of square brackets and define the expression for the values you want to store in the new `list`. In this example, the expression is `value**2`
- Then, write a `for` loop to generate the numbers you want to feed into the expression and close the square brackets. In this example, the `for` loop iterates value in `range(1, 11)`, which feeds the values 1 through 10 into the expression `value**2`.

Note that no colon is used at the end of the `for` statement.

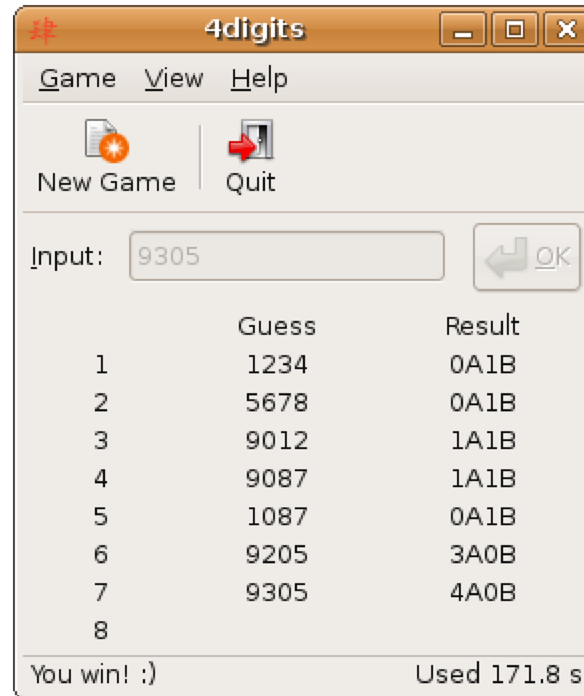
Another common operation is filtering elements to select only those that match a condition. This typically produces a `list` with fewer elements than the data being filtered. To do this in a `list` comprehension, use the `if` clause. The following includes in `list1` only the even values produced by the `for` clause:

Another common operation is filtering elements to select only those that match a condition. This typically produces a `list` with fewer elements than the data being filtered. To do this in a `list` comprehension, use the `if` clause. The following includes in `list1` only the even values produced by the `for` clause:

In [49]:
```
list1 = [item for item in range(1, 11) if item % 2 == 0]
list1
```

Out[49]:
```
[2, 4, 6, 8, 10]
```

Exercise 1: In this exercise, you will implement the Bulls and Cows game where the computer generates a random 4-digit secret number with distinct digits, and the player tries to guess it; for each guess, the program compares the input to the secret number and returns a result in the format "XAXB," where each "A" indicates a digit that is both correct and in the right position (a bull) and each "B" indicates a correct digit in the wrong position (a cow)—for instance, if the secret is 4271 and the guess is 1234, the output should be "1A2B" because the digit "2" is correctly placed while "4" and "1" are present but misplaced.



source: https://en.wikipedia.org/wiki/Bulls_and_Cows

In [ ]:

```
import random

# Generate a random four-digit number
def generate_number():
    digits = list(range(10))
    random.shuffle(digits) # randomly shuffle the list!
    return digits[:4]

# Check the user's guess against the secret number
def check_guess(guess, secret):
    # Note that both guess and secret are lists!
    a = 0 # number of correct digits in the correct position
    b = 0 # number of correct digits in the wrong position
    for _________________:  # iterate over list and get the index
        if guess[i] == secret[i]:
            a += 1
        elif __________: # Use operator to determine whether the digit is in .
            b += 1
    return a, b
```

In [ ]:

```
# Play the game
print("Welcome to 1A2B!")
print("I'm thinking of a four-digit number. Can you guess it?")
secret = generate_number()
guesses = 0
while True:
    guess = input("Enter your guess, enter 'quit' to give up: ")
    if guess == 'quit':
        print("The secret number is", secret)
        break
    elif len(guess) != 4 or not guess.isdigit():
        print("Invalid guess. Please enter a four-digit number.")
        continue
    guess = _______ # Use list comprehension to get the 4-digit guess list
    guesses += 1
    result = check_guess(guess, secret)
    print(result[0],'A', result[1], 'B', sep="")
    if result[0] == 4:
        print("Congratulations, you guessed the number in", guesses, "guesses
        break
```

Sequence Data Types

`Lists` aren’t the only data types that represent ordered sequences of values. For example, `strings` and `lists` are similar if you consider **a string to be a "list" of single text characters**.

`Lists` aren't the only data types that represent ordered sequences of values. For example, `strings` and `lists` are 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; and using them with for loops, with `len()`, and with the `in` and `not in` operators.

`Lists` aren't the only data types that represent ordered sequences of values. For example, `strings` and `lists` are 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; and using them with for loops, with `len()`, and with the `in` and `not in` operators.

In [50]:
```
'a' in 'apple'
```

Out[50]:
```
True
```

Mutable and Immutable Data Types

But `lists` and `strings` are different in an important way. A list object is a ***mutable*** data type: it can have elements 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:

But `lists` and `strings` are different in an important way. A list object is a ***mutable*** data type: it can have elements 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:

In [51]:
```
name = 'Zophie a cat'
name[7] = 't'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5960\2958416556.py in <module>
      1 name = 'Zophie a cat'
----> 2 name[7] = 't'

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

Tuples

A `tuple` is a sequence of values much like a `list`. The values stored in a `tuple` can be any type, and they are indexed by integers. The important difference is that `tuples` are ***immutable***.

A `tuple` is a sequence of values much like a `list`. The values stored in a `tuple` can be any type, and they are indexed by integers. The important difference is that `tuples` are ***immutable***.

Although it is not necessary, it is common to enclose `tuples` in parentheses to help us quickly identify `tuples` when we look at `Python` code:

A `tuple` is a sequence of values much like a `list`. The values stored in a `tuple` can be any type, and they are indexed by integers. The important difference is that `tuples` are ***immutable***.

Although it is not necessary, it is common to enclose `tuples` in parentheses to help us quickly identify `tuples` when we look at `Python` code:

In [52]:
```
type(())
```

Out[52]:
```
tuple
```

In [53]:

```
t = ('a', 'b', 'c', 'd', 'e')
type(t)
```

Out[53]:

```
tuple
```

In [53]:
```
t = ('a', 'b', 'c', 'd', 'e')
type(t)
```

Out[53]:
```
tuple
```

To create a `tuple` with a single element, you have to include the final comma or use the `tuple()` function:

```
In [53]: t = ('a', 'b', 'c', 'd', 'e')
         type(t)
```

```
Out[53]: tuple
```

To create a `tuple` with a single element, you have to include the final comma or use the `tuple()` function:

```
In [54]: t1 = ('a',)
         t2 = tuple('a')
         print(type(t1), type(t2))
         t3 = ('a')
         print(type(t3))
         print(t1, t2, t3)
```

```
<class 'tuple'> <class 'tuple'>
<class 'str'>
('a',) ('a',) a
```

If the argument of `tuple()` is a sequence (`string`, `list`, or `tuple`), the result is a `tuple` with the elements of the sequence:

If the argument of `tuple()` is a sequence (`string`, `list`, or `tuple`), the result is a `tuple` with the elements of the sequence:

In [55]:
```
t = tuple('nsysu')
t
```

Out[55]:
```
('n', 's', 'y', 's', 'u')
```

If the argument of `tuple()` is a sequence (`string`, `list`, or `tuple`), the result is a `tuple` with the elements of the sequence:

In [55]:
```
t = tuple('nsysu')
t
```

Out[55]:
```
('n', 's', 'y', 's', 'u')
```

Most `list` operators also work on `tuples`. The bracket operator indexes an element:

If the argument of `tuple()` is a sequence (`string`, `list`, or `tuple`), the result is a `tuple` with the elements of the sequence:

In [55]:
```
t = tuple('nsysu')
t
```

Out[55]:
```
('n', 's', 'y', 's', 'u')
```

Most `list` operators also work on `tuples`. The bracket operator indexes an element:

In [56]:
```
print(t[0]) # t = tuple('nsysu')
print(t[1:3])
```

```
n
('s', 'y')
```

But if you try to modify one of the elements of the `tuple`, you get an error:

But if you try to modify one of the elements of the `tuple`, you get an error:

```
In [57]: t[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5960\3054271853.py in <module>
----> 1 t[0] = 'A'

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

But if you try to modify one of the elements of the `tuple`, you get an error:

In [57]:

```
t[0] = 'A'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_5960\3054271853.py in <module>
----> 1 t[0] = 'A'

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

You can use `tuples` to convey to anyone reading your code that you don't intend for that sequence of values to change. Use a `tuple` if you need an ordered sequence of values that never changes.

In [58]:
```
display_quiz(path+"tuple.json", max_width=800)
```

Which of the following statements about lists and tuples in Python are true? (Select all that apply)

- Lists can store elements of different data types.
- The size of a tuple can be changed after it is created.
- Tuples are mutable, meaning their elements can be modified after creation.
- Lists have methods like append() and extend(), while tuples do not support these methods.
- Both lists and tuples support indexing and slicing operations.

Unpacking Sequences

We have seen the multiple assignment trick in the previous chapter (which is actually unpacking the `tuple`). In fact, you can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables.

We have seen the multiple assignment trick in the previous chapter (which is actually unpacking the `tuple`). In fact, you can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables.

In [59]:
```
student_tuple = ('Amanda', [98, 85, 87])
```

We have seen the multiple assignment trick in the previous chapter (which is actually unpacking the `tuple`). In fact, you can unpack any sequence's elements by assigning the sequence to a comma-separated list of variables.

In [59]:
```
student_tuple = ('Amanda', [98, 85, 87])
```

In [60]:
```
first_name, grades = student_tuple
print(first_name, grades)
```

```
Amanda [98, 85, 87]
```

Unpacking is widely used to return multiple values in a function:

Unpacking is widely used to return multiple values in a function:

In [61]:

```
def total_ave(grade):
    total = sum(grade)
    ave = total/len(grade)
    return total, ave

grades = [85, 70, 100, 90]
total, ave = total_ave(grades)

print(total, ave)
```

```
345 86.25
```

References

Technically, in `Python`, variables store ***references*** to the computer memory locations where the values are stored.

Technically, in `Python`, variables store ***references*** to the computer memory locations where the values are stored.

In [62]:
```
spam = 42
cheese = spam
print(id(cheese), id(spam))
spam = 100
print(id(cheese), id(spam))

spam, cheese
```

```
1951386922576 1951386922576
1951386922576 1951387112912
```

Out[62]:
```
(100, 42)
```

Technically, in `Python`, variables store ***references*** to the computer memory locations where the values are stored.

In [62]:
```
spam = 42
cheese = spam
print(id(cheese), id(spam))
spam = 100
print(id(cheese), id(spam))

spam, cheese
```

```
1951386922576 1951386922576
1951386922576 1951387112912
```

Out[62]:
```
(100, 42)
```

The identifier returned by `id()` is actually the memory address of the object, represented as a Python integer. All values in Python have a unique identity (address) that can be obtained with the `id()` function.

But `lists` don't work this way, because `list` are mutable:

But `lists` don’t work this way, because `list` are mutable:

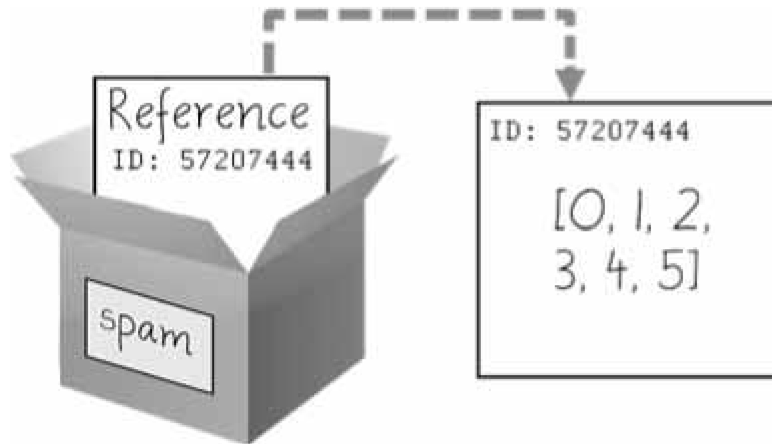
In [63]:
```
spam = [0, 1, 2, 3, 4, 5]
cheese = spam         # The reference is being copied, not the list.
print(id(cheese), id(spam))
cheese[1] = 'Hello!' # This changes the list value!
print(id(cheese), id(spam))

spam, cheese
```

```
1951470862016 1951470862016
1951470862016 1951470862016
```

Out[63]:
```
([0, 'Hello!', 2, 3, 4, 5], [0, 'Hello!', 2, 3, 4, 5])
```

Using boxes as a metaphor for variables, the following shows what happens when a `list` is assigned to the `spam` variable.

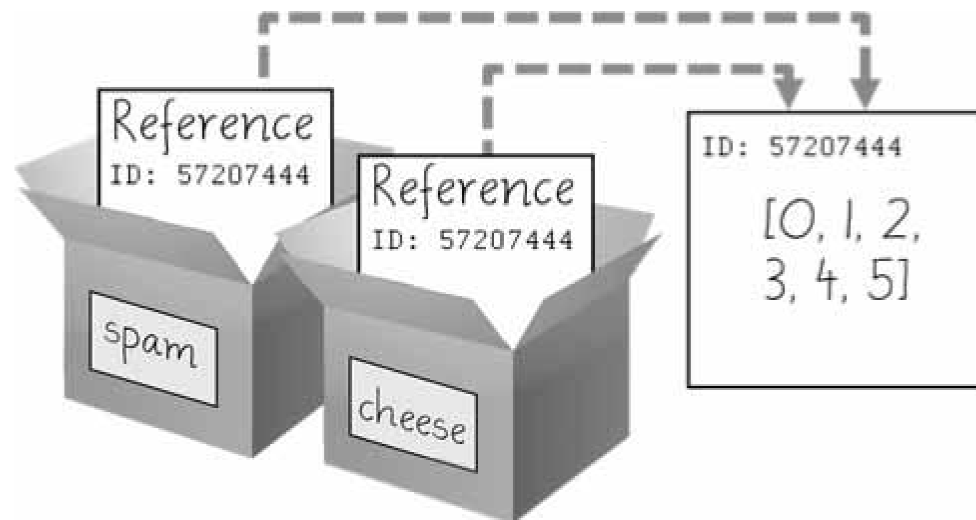
Using boxes as a metaphor for variables, the following shows what happens when a `list` is assigned to the `spam` variable.



source: https://automatetheboringstuff.com/2e/chapter4/

Then, the reference in `spam` is copied to `cheese`. Only a new reference was created and stored in `cheese`, not a new `list`. Note how both references refer to the same `list`.

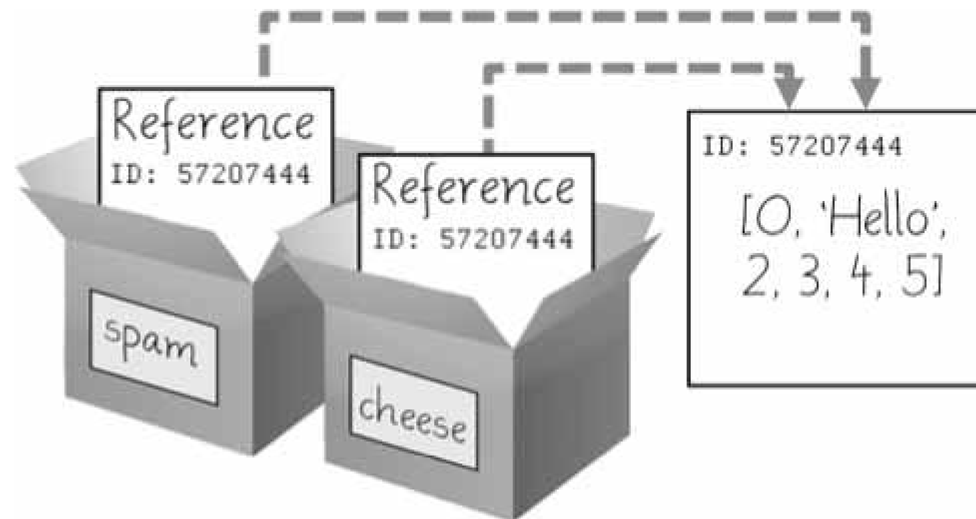
Then, the reference in `spam` is copied to `cheese`. Only a new reference was created and stored in `cheese`, not a new `list`. Note how both references refer to the same `list`.



source: https://automatetheboringstuff.com/2e/chapter4/

When you alter the `list` that `cheese` refers to, the `list` that `spam` refers to is also changed, because both `cheese` and `spam` refer to the same `list`.

When you alter the `list` that `cheese` refers to, the `list` that `spam` refers to is also changed, because both `cheese` and `spam` refer to the same `list`.



source: https://automatetheboringstuff.com/2e/chapter4/

Like `integer`, `'Hello'` is a `string` which 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`.

Like `integer`, `'Hello'` is a `string` which 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`.

In [64]:

```
bacon = 'Hello'
print(id(bacon))
bacon = bacon + 'World'
print(id(bacon))
```

```
1951470839536
1951470840624
```

However, `lists` can be modified because they are mutable objects. The `append()` method doesn’t create a new `list` object; it changes the existing `list` object. We call this **"modifying the object in-place."**

However, `lists` can be modified because they are mutable objects. The `append()` method doesn’t create a new `list` object; it changes the existing `list` object. We call this **"modifying the object in-place."**

In [65]:
```
eggs = ['Hello'] # This creates a new list.
print(id(eggs))
eggs.append('World')  # append() modifies the list "in place".
print(id(eggs))       # eggs still refers to the same list as before.
```

```
1951470861376
1951470861376
```

However, `lists` can be modified because they are mutable objects. The `append()` method doesn't create a new `list` object; it changes the existing `list` object. We call this **"modifying the object in-place."**

In [65]:

```
eggs = ['Hello'] # This creates a new list.
print(id(eggs))
eggs.append('World')  # append() modifies the list "in place".
print(id(eggs))       # eggs still refers to the same list as before.
```

```
1951470861376
1951470861376
```

If two variables refer to the same `list` (like `spam` and `cheese` in the previous section) and the `list` itself changes, both variables are affected because they both refer to the same `list`. The `append()`, `remove()`, `sort()`, and other `list` methods modify their `lists` in place.

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**.

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`, which we will describe in the next chapter), this means a **copy of the reference** is used for the parameter.

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`, which we will describe in the next chapter), this means a **copy of the reference** is used for the parameter.

In [66]:
```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

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

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`, which we will describe in the next chapter), this means a **copy of the reference** is used for the parameter.

In [66]:

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

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

Notice 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. Even though `spam` and `someParameter` contain separate references, they both refer to the same `list`.

For immutable types `string` and `integers`, we will create a new object in the function when we modify `someParameter`. Therefore, the original value will not be modified after the loop.

For immutable types `string` and `integers`, we will create a new object in the function when we modify `someParameter`. Therefore, the original value will not be modified after the loop.

In [67]:

```
def eggs(someParameter):
    print(id(someParameter))
    someParameter = someParameter + "world" # This will create a new object a
    print(id(someParameter))

spam = "hello"
print(id(spam))
eggs(spam)
print(spam)
```

```
1951465911152
1951465911152
1951470818352
hello
```

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

`Python` provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. `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.

`Python` provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. `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.

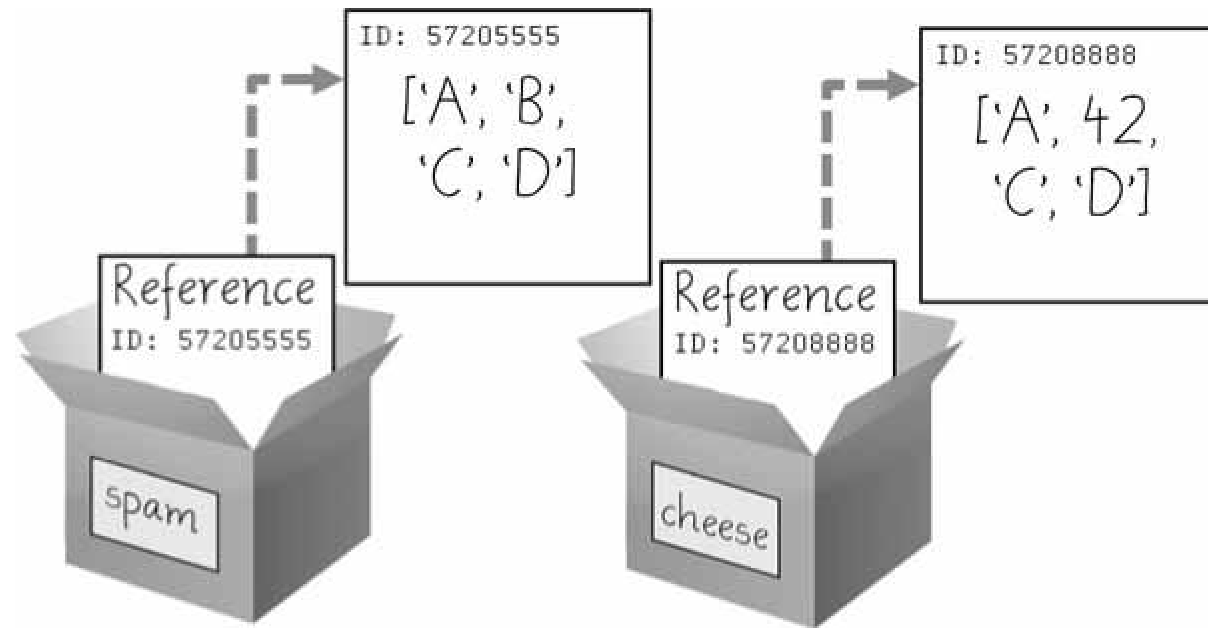
In [68]:
```
import copy

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

spam, cheese
```

```
1951469880512
1951470840320
```

Out[68]:
```
(['A', 'B', 'C', 'D'], ['A', 42, 'C', 'D'])
```

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.

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.



source: https://automatetheboringstuff.com/2e/chapter4/

Exercise 2: Here, we will simulate the process of a simple card game. The game is played with a standard deck of 52 cards, and we will randomly select 40 cards and divide them evenly between two players. Each player gets a hand of 20 cards. The goal of the game is to collect pairs of cards with the same rank (e.g., two aces, two kings, etc.). The player with the most pairs at the end of the game wins.

In [ ]:

```
import random

# Write a function create_deck that creates a list of tuples representing a s
# Each tuple should contain two elements: the rank (e.g., "ace", "king", etc.,
# and the suit (e.g., "hearts", "spades", etc.).
def create_deck():
    ranks = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"
    suits = ['♣', '♦', '♥', '♠']
    deck = [(rank, suit) ______] # Use list comprehension to create the deck.
    return deck

# A function that takes the deck as a parameter and returns two lists, each c
# cards from the deck. Use list slicing and the random module to implement th
def deal_cards(deck):
    deck = deck[:40]
    random.shuffle(deck)
    hand1 = _____  # Split it into 20 cards in each using slice
    hand2 = _____
    return hand1, hand2
```

```
In [ ]: # Write a function find_pairs that takes a list of cards as a parameter and r
        # representing the pairs of cards in the list. A pair is defined as two cards
        def find_pairs(cards):
            pairs = []
            for i, card1 in enumerate(cards):
                for j, card2 in enumerate(cards):
                    if i != j and card1[0] == card2[0] and card1 not in [pair[0] for
                        and card1 not in [pair[1] for pair in pairs] and card2 not in
                        and card2 not in [pair[1] for pair in pairs]:
                        pairs._____((card1, card2)) # Use a method from the list to a
            return pairs
```

In [ ]:

```
deck = create_deck()
hand1, hand2 = deal_cards(deck)
pairs1 = find_pairs(hand1)
pairs2 = find_pairs(hand2)

print(pairs1)
print(pairs2)
if ___________: # Compare the length of the two lists
    print("Player 1 wins!")
elif _____________:
    print("Player 2 wins!")
else:
    print("It's a tie!")
```

In [69]:

```
from jupytercards import display_flashcards
fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
display_flashcards(fpath + 'ch4.json')
```

containers

Next

>