Loop Lists

Loop Through a List

You can loop through the list items by using a `for` loop:

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]

for x in thislist:

  print(x)
```

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable.

Print all items by referring to their index number:

```
thislist = ["apple", "banana", "cherry"]


for i in range(len(thislist)):


  print(thislist[i])
```

The iterable created in the example above is `[0, 1, 2]`.

Using a While Loop

You can loop through the list items by using a `while` loop.

Use the `len()` function to determine the list length. Then, start at 0 and loop your way through the list items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Print all items, using a `while` loop to go through all the index numbers

```
thislist = ["apple", "banana", "cherry"]

i = 0

while i < len(thislist):

  print(thislist[i])

  i = i + 1
```

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

```
thislist = ["apple", "banana", "cherry"]

[print(x) for x in thislist]
```

List Comprehension

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

You want a new list based on a list of fruits, containing only the fruits with the letter "a" in the name.

Without list comprehension, you will have to write a `for` statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = []

for x in fruits:

  if "a" in x:

    newlist.append(x)

print(newlist)
```

Condition

The *condition* is like a filter that only accepts the items that valuate to `True`.

Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

The condition `if x != "apple"` will return `True` for all elements other than "apple", making the new list contain all fruits except "apple".

The *condition* is optional and can be omitted:

With no `if` statement:

```
newlist = [x for x in fruits]
```

Iterable

The *iterable* can be any iterable object, like a list, tuple, set etc.

```
newlist = [x for x in range(10)]
```

Accept only numbers lower than 5:

```
newlist = [x for x in range(10) if x < 5]
```

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

Set all values in the new list to 'hello':

```
newlist = ['hello' for x in fruits]
```

he *expression* can also contain conditions, not like a filter, but as a way to manipulate the outcome:

Return "orange" instead of "banana":

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

Sort Lists

Sort List Alphanumerically

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

Sort the list alphabetically:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]

thislist.sort()

print(thislist)
```

Sort the list numerically:

```
thislist = [100, 50, 65, 82, 23]

thislist.sort()

print(thislist)
```

Sort Descending

To sort descending, use the keyword argument `reverse = True`:

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]

thislist.sort(reverse = True)

print(thislist)
```

Sort the list descending:

```
thislist = [100, 50, 65, 82, 23]

thislist.sort(reverse = True)

print(thislist)
```

Case Insensitive Sort

By default the `sort()` method is case sensitive, resulting in all capital letters being sorted before lower case letters:

Case sensitive sorting can give an unexpected result:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]

thislist.sort()

print(thislist)
```

Luckily we can use built-in functions as key functions when sorting a list.

So if you want a case-insensitive sort function, use str.lower as a key function:

Perform a case-insensitive sort of the list:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]

thislist.sort(key = str.lower)

print(thislist)
```

Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?

The `reverse()` method reverses the current sorting order of the elements.

Reverse the order of the list items:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]

thislist.reverse()

print(thislist)
```

Copy Lists

Copy a List

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

Use the copy() method

You can use the built-in List method `copy()` to copy a list.

Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]

mylist = thislist.copy()

print(mylist)
```

Use the list() method

Another way to make a copy is to use the built-in method `list()`.

Make a copy of a list with the `list()` method:

```
thislist = ["apple", "banana", "cherry"]

mylist = list(thislist)

print(mylist)
```

Use the slice Operator

You can also make a copy of a list by using the `:` (slice) operator.

Make a copy of a list with the `:` operator:

```
thislist = ["apple", "banana", "cherry"]

mylist = thislist[:]

print(mylist)
```

Join Lists

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the `+` operator.

Join two list:

```
list1 = ["a", "b", "c"]

list2 = [1, 2, 3]

list3 = list1 + list2

print(list3)
```

Another way to join two lists is by appending all the items from list2 into list1, one by one:

Append list2 into list1:

```
list1 = ["a", "b" , "c"]

list2 = [1, 2, 3]


for x in list2:

  list1.append(x)


print(list1)
```

Or you can use the `extend()` method, where the purpose is to add elements from one list to another list:

Use the `extend()` method to add list2 at the end of list1:

```
list1 = ["a", "b" , "c"]

list2 = [1, 2, 3]

list1.extend(list2)

print(list1)
```

List Methods

List Methods

Python has a set of built-in methods that you can use on lists.

```
# Example list

fruits = ["apple", "banana", "cherry"]


# append() - Adds an element at the end of the list

fruits.append("orange")

print(fruits)  # Output: ['apple', 'banana', 'cherry',
'orange']
```

```
# clear() - Removes all the elements from the list

fruits.clear()

print(fruits)  # Output: []


# Resetting the list for further examples

fruits = ["apple", "banana", "cherry"]


# copy() - Returns a copy of the list

fruits_copy = fruits.copy()

print(fruits_copy)  # Output: ['apple', 'banana', 'cherry']


# count() - Returns the number of elements with the specified
value

print(fruits.count("banana"))  # Output: 1


# extend() - Adds the elements of a list (or any iterable) to
the end of the current list

fruits.extend(["grape", "kiwi"])

print(fruits)  # Output: ['apple', 'banana', 'cherry',
'grape', 'kiwi']


# index() - Returns the index of the first element with the
specified value

print(fruits.index("cherry"))  # Output: 2


# insert() - Adds an element at the specified position

fruits.insert(1, "mango")

print(fruits)  # Output: ['apple', 'mango', 'banana',
'cherry', 'grape', 'kiwi']
```

```
# pop() - Removes the element at the specified position

fruits.pop(3)

print(fruits)  # Output: ['apple', 'mango', 'banana',
'grape', 'kiwi']


# remove() - Removes the item with the specified value

fruits.remove("banana")

print(fruits)  # Output: ['apple', 'mango', 'grape', 'kiwi']


# reverse() - Reverses the order of the list

fruits.reverse()

print(fruits)  # Output: ['kiwi', 'grape', 'mango', 'apple']


# sort() - Sorts the list

fruits.sort()

print(fruits)  # Output: ['apple', 'grape', 'kiwi', 'mango']
```

Unpack Tuples

Unpacking a Tuple

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

Packing a tuple:

```
fruits = ("apple", "banana", "cherry")
```

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)

print(yellow)

print(red)
```

Using Asterisk*

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list:

Assign the rest of the values as a list called "red":

```
fruits = ("apple", "banana", "cherry", "strawberry",
"raspberry")

(green, yellow, *red) = fruits

print(green)

print(yellow)

print(red)
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Add a list of values the "tropic" variable:

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")

(green, *tropic, red) = fruits

print(green)

print(tropic)

print(red)
```

Loop Tuples

Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")

for x in thistuple:

  print(x)
```

Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable.

Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")

for i in range(len(thistuple)):

  print(thistuple[i])
```

Using a While Loop

You can loop through the tuple items by using a `while` loop.

Use the `len()` function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Print all items, using a `while` loop to go through all the index numbers:

```
thistuple = ("apple", "banana", "cherry")

i = 0

while i < len(thistuple):

  print(thistuple[i])

  i = i + 1
```

Join Tuples

Join Two Tuples

To join two or more tuples you can use the `+` operator:

Join two tuples:

```
tuple1 = ("a", "b" , "c")

tuple2 = (1, 2, 3)


tuple3 = tuple1 + tuple2

print(tuple3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the `*` operator:

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")

mytuple = fruits * 2

print(mytuple)
```

Tuple Methods

```
# Example tuple
my_tuple = (1, 2, 3, 2, 4, 2, 5)

# count() - Returns the number of times a specified value occurs in a tuple
occurrences = my_tuple.count(2)
print("Count of 2:", occurrences)  # Output: Count of 2: 3

# index() - Searches the tuple for a specified value and returns the position where it was found
position = my_tuple.index(3)
print("Index of 3:", position)  # Output: Index of 3: 2
```

While Loops

Python Loops

Python has two primitive loop commands:

- `while` loops
- `for` loops

The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

Print i as long as i is less than 6:

```
i = 1

while i < 6:

  print(i)

  i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Exit the loop when i is 3:

```
i = 1

while i < 6:

  print(i)

  if i == 3:

    break

  i += 1
```

The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The else Statement

With the `else` statement we can run a block of code once when the condition no longer is true:

```
i = 1

while i < 6:

  print(i)

  i += 1

else:

  print("i is no longer less than 6")
```

Loops

Python For Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)
```

The `for` loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Loop through the letters in the word "banana":

```
for x in "banana":

  print(x)
```

The break Statement

With the `break` statement, we can stop the loop before it has looped through all the items:

Exit the loop when `x` is "banana":

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)

  if x == "banana":

    break
```

Exit the loop when `x` is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  if x == "banana":

    break

  print(x)
```

The continue Statement

With the `continue` statement we can stop the current iteration of the loop, and continue with the next:

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  if x == "banana":

    continue

  print(x)
```

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Note: The `else` block will NOT be executed if the loop is stopped by a `break` statement.

Break the loop when `x` is 3, and see what happens with the `else` block:

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```

The pass Statement

`for` loops cannot be empty, but if you for some reason have a `for` loop with no content, put in the `pass` statement to avoid getting an error.

```
for x in [0, 1, 2]:
  pass
```