Conditional Statements & Lists

September 21, 2023

1 Conditional Statements

Conditional statements in Python are used to execute different blocks of code based on specified conditions. They allow your program to make decisions and choose different paths of execution, depending on whether certain conditions are met. The most common types of conditional statements in Python are if, elif (short for "else if"), and else. Here's how they work:

- If Else
- If
- Else
- El If (else if)
- If Else Ternary Expression

1.0.1 IF - Statement

1 - if Statement: The if statement is used to execute a block of code only if a specified condition is True. If the condition is False, the code block is skipped.

if condition: # Code to execute if the condition is True

[1]:
```
x = 10
if x > 5:
    print("x is greater than 5")
```

```
x is greater than 5
```

1.0.2 If Else Statement

2 - if...else Statement: The if...else statement allows you to execute one block of code if a condition is True, and another block if the condition is False.

if condition: # Code to execute if the condition is True else: # Code to execute if the condition is False

[2]:
```
age = 18
if age >= 18:
    print("you are an adult.")
else:
    print("you are not an adult.")
```

```
you are an adult.
```

1.0.3 If, elif, else - Statement

3 - if…elif…else Statement: The if…elif…else statement allows you to test multiple conditions in sequence. If the first condition is True, its code block is executed. If it's False, the next elif condition is tested, and so on. If none of the conditions is True, the else block (if provided) is executed.

if condition1: # Code to execute if condition1 is True elif condition2: # Code to execute if condition2 is True else: # Code to execute if no conditions are True

[3]:
```
grade = 85
if grade >= 90:
    print("A")
elif grade >= 80:
    print("B")
elif grade >= 70:
    print("C")
else:
    print("F")
```

```
B
```

These conditional statements help you write programs that can respond dynamically to different situations. You can also nest if statements within other if statements to create more complex decision-making structures. Conditional statements are fundamental to control the flow of your Python programs and make them capable of performing a wide range of tasks.

2 LISTS

- List Basics
- List Operations
- List Comprehensions / Slicing
- List Methods

2.1 List Basics

[4]:
```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

[5]:
```
# Note: There are some list methods that will change the order, but in general:␣
↪the order of the items will not change.
# Allow Duplicates
# Since lists are indexed, lists can have items with the same value:

thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

[6]:
```
# List Length
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

```
3
```

[7]:
```
# List Items - Data Types
# List items can be of any data type:


list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]

print(list1)
print(list2)
print(list3)
```

```
['apple', 'banana', 'cherry']
[1, 5, 7, 9, 3]
[True, False, False]
```

[8]:
```
# A list can contain different data types:
# A list with strings, integers and boolean values:

list1 = ["abc", 34, True, 40, "male"]
print(list1)
```

```
['abc', 34, True, 40, 'male']
```

[9]:
```
# type()
# From Python's perspective, lists are defined as objects with the data type␣
 ↪'list':

mylist = ["apple", "banana", "cherry"]
print(type(mylist))
```

```
<class 'list'>
```

[10]:
```
# The list() Constructor
# It is also possible to use the list() constructor when creating a new list.

thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
```

```
['apple', 'banana', 'cherry']
```

Python Collections (Arrays) There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- Dictionary is a collection which is ordered** and changeable. No duplicate members.

2.2 List Operations

Lists in Python support a variety of operations that allow you to manipulate and work with the data they contain.

[11]:
```
# 1 - Concatenation (+): You can concatenate (combine) two or more lists to␣
↪create a new list.

list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2  # [1, 2, 3, 4, 5, 6]

print(result)
```

```
[1, 2, 3, 4, 5, 6]
```

[12]:
```
# 2- Repetition (*): You can repeat a list by multiplying it with an integer.

orginal_list = [1, 2]
repeated_list = orginal_list * 3    # [1, 2, 1, 2, 1, 2]

print(repeated_list)
```

```
[1, 2, 1, 2, 1, 2]
```

[13]:
```
# 3 - Membership (in and not in): You can check if an item is present in a list␣
↪using the in and not in operators.

my_list = [1, 2, 3, 4, 5]
3 in my_list   #True
6 not in my_list   #True
```

[13]:
```
True
```

[17]:
```
# 4 - List Length (len()): You can find the number of elements in a list using␣
↪the len() function.

my_list = [1, 2, 3, 4, 5]
length = len(my_list)  # 5
print(length)
```

```
5
```

[19]:
```
# 5 - List Iteration (for loop): You can iterate through the elements of a list␣
↪using a for loop.

my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)
```

```
1
2
3
4
5
```

[20]:
```
# 6 - List Slicing: You can extract a portion of a list using slicing.

my_list = [1, 2, 3, 4, 5]
subset = my_list[1:4]        #[2, 3, 4]
print(subset)
```

```
[2, 3, 4]
```

[21]:
```
# 7 - Sorting (sorted() method): You can sort the elements of a list in␣
↪ascending order using the sorted() method.

my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_list = sorted(my_list)     # [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

print(sorted_list)
```

```
[1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

[25]:
```
# 8 - List Methods: Lists have various built-in methods, such as append(),␣
↪insert(), remove(), pop(), and extend(), which allow you to manipulate list␣
↪elements in different ways.

my_list = [1, 2, 3]
my_list.append(4)   # [1, 2, 3, 4]
my_list.insert(1, 5)   # [1, 5, 2, 3, 4]
my_list.remove(2)    # [1, 5, 3, 4]
item = my_list.pop()     # Removes and returns the last item (4), my_list␣
↪becomes [1, 5, 3]
print(item)
```

```
4
```

[26]:
```
# 9 - List Comprehensions: List comprehensions provide a concise way to create␣
↪new lists by applying an expression to each item in an existing list.
numbers = [1, 2, 3, 4, 5]
squared_numbers = [x ** 2 for x in numbers]  # [1, 4, 9, 16, 25]

print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

2.3 List Comprehensions / Slicing

List comprehensions and slicing are two powerful features in Python for working with lists. They provide concise and efficient ways to create new lists or extract subsets of existing lists. ###### List Comprehensions:

A list comprehension is a concise way to create a new list by applying an expression to each item in an existing iterable (e.g., a list) and optionally filtering the items based on a condition. The syntax for a list comprehension is as follows:

- new_list = [expression for item in iterable if condition]

1. expression: The operation to perform on each item.
2. item: A variable that represents each item in the iterable.
3. iterable: The original iterable (e.g., a list) to iterate over.
4. condition (optional): A condition that filters items (only items where the condition is True are included in the new list).

[27]:
```
# Example 1: Squaring Numbers in a List Using List Comprehension:
numbers = [1, 2, 3, 4, 5]
squared_numbers = [x ** 2 for x in numbers]
print(squared_numbers)


# Result: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

[28]:
```
# Example 2: Filtering Odd Numbers Using List Comprehension:
numbers = [1, 2, 3, 4, 5]
odd_numbers = [x for x in numbers if x % 2 != 0]
print(odd_numbers)
```

```
[1, 3, 5]
```

2.3.1 Slicing:

Slicing is a way to extract a portion (sublist) of a list. It's done by specifying a range of indices using the colon : operator. The basic syntax for slicing is:

- sublist = my_list[start:stop:step]

1. start (optional): The index where the slice starts (inclusive).
2. stop (optional): The index where the slice ends (exclusive).
3. step (optional): The step size between elements in the slice (defaults to 1).

[29]:
```
# Example 1: Slicing a List:
my_list = [1, 2, 3, 4, 5]
subset = my_list[1:4]
print(subset)
```

```
[2, 3, 4]
```

[30]:
```
# Example 2: Slicing with a Step:
my_list = [1, 2, 3, 4, 5]
subset = my_list[1:5:2]
print(subset)
```

```
[2, 4]
```

List comprehensions and slicing are essential tools for working with lists efficiently and effectively in Python. They are often used in data processing, filtering, and transforming data stored in lists.

2.4 List Methods

lists come with several built-in methods that allow you to manipulate and work with the elements of lists

[32]:
```
# append(item): Adds an item to the end of the list.
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
```

```
[1, 2, 3, 4]
```

[33]:
```
# insert(index, item): Inserts an item at a specific index within the list.
my_list = [1, 2, 3]
my_list.insert(1, 5)
print(my_list)
```

```
[1, 5, 2, 3]
```

[34]:
```
# remove(item): Removes the first occurrence of the specified item from the␣
  ↪list.
my_list = [1, 2, 3, 2]
my_list.remove(2)
print(my_list)
```

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

[37]:
```
# pop(index): Removes and returns the item at the specified index. If no index␣
↪is provided, it removes and returns the last item.
my_list = [1, 2, 3]
popped_item = my_list.pop(1)
print(popped_item)
print(my_list)
```

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

[38]:
```
# extend(iterable): Appends the elements of an iterable (e.g., a list, tuple)␣
↪to the end of the list.
my_list = [1, 2, 3]
my_list.extend([4, 5])
print(my_list)
```

```
[1, 2, 3, 4, 5]
```

[40]:
```
# index(item): Returns the index of the first occurrence of the specified item.
my_list = [1, 2, 3, 2]
index = my_list.index(2)
print(index)
```

```
1
```

[41]:
```
# count(item): Returns the number of times the specified item appears in the␣
↪list.
my_list = [1, 2, 3, 2]
count = my_list.count(2)
print(count)
```

```
2
```

[42]:
```
# reverse(): Reverses the order of elements in the list.
my_list = [1, 2, 3]
my_list.reverse()
print(my_list)
```

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

[43]:
```
# sort(): Sorts the elements of the list in ascending order. You can use the␣
↪reverse=True argument to sort in descending order.
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
my_list.sort()
# Result: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
print(my_list)
```

```
[1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

[44]:
```
# clear(): Removes all elements from the list, making it empty.
my_list = [1, 2, 3]
my_list.clear()
print(my_list)
```

```
[]
```

These list methods provide a wide range of functionality for working with lists in Python, making it easier to manipulate, search, and modify list elements according to your needs.

[45]:
```
# The copy() method returns a copy of the specified list.
fruits = ['apple', 'banana', 'cherry', 'orange']

x = fruits.copy()
print(x)
```

```
['apple', 'banana', 'cherry', 'orange']
```

3 PRACTICE

[49]:
```
#  Access List Items
# List items are indexed and you can access them by referring to the index␣
 ↪number:
thislist = ["apple", "banana", "cherry"]
print(thislist[1])

# Note: The first item has index 0.
```

```
banana
```

[50]:
```
# Negative Indexing
# Negative indexing means start from the end
# -1 refers to the last item, -2 refers to the second last item etc.

thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

```
cherry
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

Note: The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

[51]:
```
# Return the third, fourth, and fifth item:

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
```

```
['cherry', 'orange', 'kiwi']
```

[52]:
```
# This example returns the items from the beginning to, but NOT including,␣
  ↪"kiwi":

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

```
['apple', 'banana', 'cherry', 'orange']
```

[53]:
```
# This example returns the items from "cherry" to the end:

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

```
['cherry', 'orange', 'kiwi', 'melon', 'mango']
```

Change List Items To change the value of a specific item, refer to the index number:

[54]:
```
# Change the second item:
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

```
['apple', 'blackcurrant', 'cherry']
```

Change a Range of Item Values To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

[55]:
```
# Change the values "banana" and "cherry" with the values "blackcurrant" and␣
  ↪"watermelon":

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

[56]:
```
# Change the second value by replacing it with two new values:
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

```
['apple', 'blackcurrant', 'watermelon', 'cherry']
```

Note: The length of the list will change when the number of items inserted does not match the number of items replaced.

If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

[57]:
```
# Change the second and third value by replacing it with one value:

thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

```
['apple', 'watermelon']
```

3.0.1 Insert Items

To insert a new list item, without replacing any of the existing values, we can use the insert() method.

The insert() method inserts an item at the specified index:

Note: As a result of the example below, the list will now contain 4 items.

[58]:
```
# Insert "watermelon" as the third item:

thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

```
['apple', 'banana', 'watermelon', 'cherry']
```

3.0.2 Remove Specified Item

The remove() method removes the specified item.

[59]:
```
# Remove "banana":

thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

```
['apple', 'cherry']
```

[60]:
```
# Remove the first occurance of "banana":

thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)
```

```
['apple', 'cherry', 'banana', 'kiwi']
```

3.0.3 Remove Specified Index

The pop() method removes the specified index.

[61]:
```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

```
['apple', 'cherry']
```

[62]:
```
# Remove the last item:

thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

```
['apple', 'banana']
```

[63]:
```
# Remove the first item:

thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```
['banana', 'cherry']
```

[64]:
```
# Delete the entire list:

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

3.0.4 Clear the List

The clear() method empties the list.

The list still remains, but it has no content.

[65]:
```
# Clear the list content:

thislist = ["apple", "banana", "cherry"]
thislist.clear()
```

```
print(thislist)
```

```
[]
```

3.0.5 Join 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.

[66]:
```
# Join two list:

list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

```
['a', 'b', 'c', 1, 2, 3]
```

[67]:
```
#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)
```

```
['a', 'b', 'c', 1, 2, 3]
```

[68]:
```
# 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)
```

```
['a', 'b', 'c', 1, 2, 3]
```

[ ]:
```

```