Internet of Things 420-420-LE

Week 5: Iteration and Functions

CHAMPLAIN COLLEGE

Rev.: Jan. 15th, 2025

Introducing the for loop

Conditionals

Conditionals are used to tell your computer to do a set of instructions depending on whether a Boolean is True or not. In other words, we are telling the computer:

```
if something is true:
    do task a
otherwise:
    do task b
```

In fact, the syntax in Python is almost the same:

```
month = 'AUG'

if month == 'AUG':
   print('This month is August.')
```

1. The Boolean expression, month == 'AUG', is called the condition. If it is True, the ***indented*** statement below it is executed.
2. Python uses the colon to act as the "then" keyword

Indentation matters...

Any lines with the same level of indentation will be evaluated together.

```
month = 'AUG'

if month == 'AUG':
   print('This month is August.')
   print('Same level of indentation, so still printed!')
```

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code

Last comment about indentation

- Although you can use either tabs or spaces for indentation, you cannot mix and match your indentation.
- For example, if you use a single tab for indentation, then stick with a single tab.
- If you use three spaces for indentation, then continue to use three spaces.
- You cannot use one space one time, one tab next time, three spaces the third time…
- It’s better to left the IDE to manage the indentation (VS Code or Thonny), **hard** to follow on a text editor. Point for the IDE!
- Be careful when copying text from internet or asking to your AI friend ;)

Nested Conditionals

We can also nest the conditionals:

```
month = 'DEC'

if month == 'JAN' or month == 'FEB' or month == 'MAR':
    print('this month belongs to Q1')
else:
    if month == 'APR' or month == 'MAY' or month == 'JUN':
        print('this month belongs to Q2')
    else:
        if month == 'AUG' or month == 'SEP' or month == 'OCT':
            print('This month belongs to Q3')
        else:
            print('This month belongs to Q4')
```

Notice that the indentation defines which clause the statement belongs to. E.g., the second if statement is executed as part of the first else clause.

elif

While this nesting is very nice, we can be more concise by using an elif clause.

```
if month == 'JAN' or month == 'FEB' or month == 'MAR':
    print('this month belongs to Q1')
elif month == 'APR' or month == 'MAY' or month == 'JUN':
    print('this month belongs to Q2')
elif month == 'AUG' or month == 'SEP' or month == 'OCT':
    print('This month belongs to Q3')
else:
    print('This month belongs to Q4')
```

Lists and tuples

SEQUENCES OF ARBITRARY OBJECTS, CALLED ITEMS

Lists

- Lists and tuples are sequences of arbitrary objects, called **items** or **elements**. They are a way to create a single object that contains many other objects
- We create lists by putting **values** or ***expressions*** inside **square brackets**, separated by **commas**:

```
my_list_1 = [1, 2, 3, 4]
type(my_list_1)
<class 'list'>
```

Although the elements of the list are integers, the type of my_list_1 is list.

- Any Python expression can be inside a list (including another list):

```
my_list_2 = [1, 2.4, 'a string', ['a string in another list', 5]]
my_list_2
[1, 2.4, 'a string', ['a string in another list', 5]]
my_list_3 = [2+3, 5*3, 4**2]
my_list_3
[5, 15, 16]
```

Lists

- We can also create a list by type conversion.
- For example, we can convert a string into a list of characters:

```
my_str = 'A string.'
list(my_str)
['A', ' ', 's', 't', 'r', 'i', 'n', 'g', '.']
```

List operators

- Operators on lists behave much like operators on strings. The + operator on lists means list concatenation

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

- The * operator on lists means list replication and concatenation.

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

Membership operators

- Membership operators are used to determine if an item is in a list. The two membership operators are:

English	operator
is a member of	in
is not a member of	not in

The result of the operator is True or False.

```
my_list_2 = [1, 2.4, 'a string', ['a string in another list', 5]]
1 in my_list_2
True
['a string in another list', 5] in my_list_2
True
'a string in another list' in my_list_2
False
```

We see that the string 'a string in another list' is not in my_list_2. This is because that string itself is not one of the four items of my_list_2. The string 'a string in another list' is in a list that is an item in my_list_2.

Membership operators

- Now, these membership operators offer a great convenience for conditionals:

```
Q1_months = ['JAN', 'FEB', 'MAR']
Q2_months = ['ABR', 'MAY', 'JUN']
Q3_months = ['JUL', 'AUG', 'SEP']

month='ABR'

if month in Q1_months:
    print('this month belongs to Q1')
elif month in Q2_months:
    print('this month belongs to Q2')
elif month in Q3_months:
    print('This month belongs to Q3')
else:
    print('This month belongs to Q4')
```

The simple expression

`month in Q1_months`

replaced the more verbose

```
month == 'JAN' or
month == 'FEB' or
month == 'MAR'
```

List indexing

- Because a list is **ordered**, we can ask for the first item, the second item, the nth item, the last item, etc.
- This is done using a bracket notation:

```
my_list = [1, 2.4, 'a string', ['a string in another list', 5]]
my_list[1]
2.4
```

- Notice that indexing in Python starts **at zero**

List indexing

- Because a list is **ordered**, we can ask for the first item, the second item, the nth item, the last item, etc.
- This is done using a bracket notation:

```
my_list = [1, 2.4, 'a string', ['a string in another list', 5]]
my_list[1]
2.4
```

- Notice that indexing in Python starts **at zero**
- We can also index the list that is within my_list by adding another set of brackets.

```
my_list[3][0]
'a string in another list'
```

List indexing

- There are more ways to specify items in a list

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
my_list[4]
4
my_list[-1]
10
```

- This is very convenient for indexing in reverse.

Values	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Forward indices	0	1	2	3	4	5	6	7	8	9	10
Reverse indices	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

List slicing

- What if we want to pull out multiple items in a list, called slicing? We can use colons (:) for that

```
my_list[0:5]
[0, 1, 2, 3, 4]
```

We got elements 0 through 4.

- When using the colon indexing, my_list[i:j], we get items i through j-1.
- The range is **inclusive of the first index and exclusive of the last**.
- If the slice's final index is larger than the length of the sequence, the slice ends at the last element.

List slicing

Values	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Forward indices	0	1	2	3	4	5	6	7	8	9	10
Reverse indices	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- We can also use negative indices with colons:

```
my_list[0:-3]
[0, 1, 2, 3, 4, 5, 6, 7]
```

Again, note that we only went to index -4.

- We can also specify a ***stride***. The stride comes after a second colon. For example, if we only wanted the even numbers, we could do the following.

```
my_list[0::2]
[0, 2, 4, 6, 8, 10]
```

Notice that we did not enter anything for the end value of the slice. If the end is left blank, the default is to include the entire string

A note on the stride

Values	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Forward indices	0	1	2	3	4	5	6	7	8	9	10
Reverse indices	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- Stride refers to how many positions to move **forward** after the first element is retrieved from the list

```
my_list[::-1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
my_list[::-2]
[10, 8, 6, 4, 2, 0]
my_list[-2::-2]
[9, 7, 5, 3, 1]
```

List slicing

- Similarly, we can leave out the start index, as its default is zero

```
my_list[::2]
[0, 2, 4, 6, 8, 10]
```

- So, in general, the indexing scheme is: `my_list[start:end:stride]`
  - If there are no colons, a single element is returned.
  - If there are any colons, we are slicing the list, and a list is returned.
  - If there is one colon, stride is assumed to be 1.
  - If start is not specified, it is assumed to be zero.
  - If end is not specified, the interpreted assumed you want the entire list.
  - If stride is not specified, it is assumed to be 1

Mutability

- Lists are ***mutable***. That means that you can change their values without creating a new list. (but you cannot change the data type or identity.)

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
my_list[3] = 'four'
my_list
[1, 2, 3, 'four', 5, 6, 7, 8, 9, 10]
```

- The other data types we have encountered so far, int, float, and str, are **immutable**. You cannot change their values without **reassigning** them

Mutability

- To see this, we'll use the id() function, which tells us where in memory that the variable is stored.

```
a = 689
id(a)
4482834064
a = 690
id(a)
4482833008
```

The identity of a changed when we tried to change its value. So, we didn't change its value; we made a new variable. With lists, though, this is not the case

```
id(my_list)
4483938112
my_list[0] = 'zero'
id(my_list)
4483938112
```

It is still the same list! This is very important to consider when we do assignments

Aliasing

- Aliasing is a subtle issue which can come up when assigning lists to variables.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
my_list_2 = my_list
my_list_2[0] = 'a'

my_list_2
['a', 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We see that assigning a list to a variable does not copy the list! Instead, you just get a new reference to the same value.

- Now, let's look at our original list to see what it looks like:

```
my_list
['a', 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Tuples

Tuples

- A tuple is just like a list, except it is immutable (basically a read-only list).
- A tuple is created just like a list, except we use parentheses instead of brackets.
- A tuple with a single item ***needs to include*** a comma after the item.

```
my_tuple = (0,)
not_a_tuple = (0) # this is just the number 0
type(my_tuple), type(not_a_tuple)
(tuple, int)
```

Conversion

- We can also create a tuple by doing a type conversion.

```
my_list = [1, 2.4, 'a string', ['a sting in another list', 5]]
my_tuple = tuple(my_list)
my_tuple
(1, 2.4, 'a string', ['a sting in another list', 5])
```

- Note that the list within my_list did not get converted to a tuple. It is still a list, and it is mutable.

```
type(my_tuple[3])
<class 'list’>

my_tuple[3][0] = 'a string in a list in a tuple’
my_tuple
(1, 2.4, 'a string', ['a string in a list in a tuple', 5])
```

Conversion

- However, if we try to change an item in a tuple, we get an error:

```
my_tuple[1] = 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- Even though the list within the tuple is mutable, we still cannot change the identity of that list.

```
my_tuple[3] = ['a', 'new', 'list']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Slicing of tuples

- Slicing of tuples is the same as lists, except a tuple is returned from the slicing operation, not a list:

```
my_tuple = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
my_tuple[::-1]
(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
# Odd numbers
my_tuple[1::2]
(1, 3, 5, 7, 9)
```

The + operator with tuples

- As with lists we can concatenate tuples with the + operator.

```
my_tuple + (11, 12, 13, 14, 15)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

my_tuple * 2
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Membership operators with tuples

- Membership operators work the same as with lists:

```
5 in my_tuple
True
'LeBron James' not in my_tuple
True
```

Tuple unpacking

- It is like a multiple assignment statement that is best seen through example.

```
my_tuple = (1, 2, 3)
(a, b, c) = my_tuple
a
1
b
2
c
3
```

This is useful when we want to return more than one value from a function and further using the values as stored in different variables.

- Note that the parentheses are dispensable:

```
a, b, c = my_tuple

print(a, b, c)
1 2 3
```

Wisdom on tuples and lists

- In practice, tuples and lists are very similar, differing essentially only in mutability.
- “When should I use a tuple and when should I use a list?”
  - ***Always use tuples instead of lists unless you need mutability.***

- This keeps you out of trouble. It is very easy to inadvertently change one list, and then another list (that is actually the same, but with a different variable name) gets corrupted.

Lab 3

CHECK MOODLE FOR INSTRUCTIONS

Declaration