Introduction to Programming W6

Lists and dictionaries II.

2022/05/13

1 Lists 2

A new element can be added to the end of a list using the `.append()` method. It takes a single argument (inside the parentheses): the new element to be added. Try the following code:

```
languages = ["FORTRAN", "COBOL", "LISP"]
print("initial list:", languages)
languages.append("ALGOL")
print("updated list:", languages)
```

If you try adding multiple elements at once, you will encounter a TypeError. For example, the following code will result in *TypeError: append() takes exactly one argument (2 given).*

```
languages = ["FORTRAN", "COBOL", "LISP"]
languages.append("ALGOL", "BASIC")
```

However, you can append a list to a list (with multiple elements):

```
languages = ["FORTRAN", "COBOL", "LISP"]
print("initial list:", languages)
languages.append(["ALGOL", "BASIC"])
print("updated list:", languages) #note that the index 3 element is a list!
```

The + operator can be used to merge two lists into one:

```
languages_1 = ["FORTRAN", "COBOL", "LISP"]
languages_2 = ["ALGOL", "SNOBOL", "BASIC"]
print("initial lists:", languages_1, languages_2)
new_list = languages_1 + languages_2
print("merged list:", new_list)
```

Using the + operator above is equivalent to using the `.extend()` list method that takes an iterable data type as its argument (e.g., list, string) and add it to the end of the list.

```
languages_1 = ["FORTRAN", "COBOL", "LISP"]
languages_2 = ["ALGOL", "SNOBOL", "BASIC"]
print("Initial languages 1 list:", languages_1)
#in this example, we don't create a new list, just extend languages 1
languages_1.extend(languages_2)
print("extended languages 1 list:", languages_1)
```

Above, the `languages_1.extend(languages_2)` line is practically equivalent to using `languages_1 = languages_1 + languages_2` or `languages_1 += languages_2`.

You can use a for loop to append elements to a list one-by-one:

```
languages_1 = ["FORTRAN", "COBOL", "LISP"]
languages_2 = ["ALGOL", "SNOBOL", "BASIC"]
print("initial list:", languages_1)
for language in languages_2:
    languages_1.append(language)
print("updated list:", languages_1) #note that the order is preserved!
```

A while loop can be used as well with the appropriate condition in the while header:

```
our_list = [] #defining an empty list
i = 0 #the variable we will increment
while len(our_list) < 10: #condition: do the iteration until the length reaches 10
    our_list.append(i) #append the current value of i to our list
    i += 1 #increment i

print("Our list:", our_list)
print("The length of our list:", len(our_list))
maximum = max(our_list) #max value
minimum = min(our_list) #min value
#making a line break using the "\n' newline character
print("maximum:", maximum, "\nminimum:", minimum) #note the newline
```

Note that `our_list` will have numbers 0 to 9, so the length is 10. The `max()` and `min()` functions are used to return the maximum and minimum value of a list, respectively. The newline character `\n` is used to make a line break between the arguments in the bottom print function (`\n` part of `"\nminimum:"`). The `.insert()` method inserts an element to a list at the index specified. The first argument is the index (an integer), and the second is the object to be inserted:

```
languages = ["FORTRAN", "COBOL", "LISP"]
languages.insert(1, "ALGOL") #insert the string "ALGOL" to index 1
print(languages)
```

The `.remove()` method removes the first matching element passed as an argument:

```
languages = ["FORTRAN", "COBOL", "LISP", "COBOL"] # here, the string COBOL is in the list twice
#the remove method will remove the first instance corresponding to the argument
languages.remove("COBOL")
print(languages) #the first COBOL string is removed
languages.remove("COBOL")
print(languages) #no "COBOL" in the list, because we run remove() twice
```

The `.pop()` method removes the element at the given index passed as an integer argument, and returns the removed item. It is not required to store the popped element in a new variable, but can be useful in certain situations.

```
languages = ["FORTRAN", "COBOL", "LISP", "ALGOL"]
print("initial list:", languages)
languages.pop(2) #LISP is removed
#at this point, our list is ['FORTRAN', 'COBOL', 'ALGOL']
print("modified list:", languages)

#with the next line, COBOL will be popped and returned to the variable popped_element
popped_element = languages.pop(1)
print("2x modified list:", languages)
#the pop method returned the removed element, that is stored in variable popped_element!
print("the removed element with pop:", popped_element)
```

The `.reverse()` method simply reverses the order of a given list.

```
languages = ["FORTRAN", "COBOL", "LISP"]
print("Initial list:", languages)
languages.reverse()
print("Reversed list:", languages)
```

The `.reverse()` does not take any arguments, and does not return anything (so *do not* assign it to a new variable like `languages_2 = languages.reverse()`).
The `.copy()` method creates an identical copy of a list. This method does not take any arguments.

```
languages_1 = ["FORTRAN", "COBOL", "LISP"]
languages_2 = languages_1.copy()
print("languages 1:", languages_1)
print("languages 2:", languages_2) #note that the content of the two lists is the same
```

It is also possible to copy a list with just assignment:

```
languages_1 = ["FORTRAN", "COBOL", "LISP"]
languages_2 = languages_1
print("languages 1:", languages_1)
print("languages 2:", languages_2)
```

However, it is very important that in this way if you modify `languages_2` (e.g., appending a new element to it), `languages_1` will be also modified, because `languages_2` is just a reference to `languages_1` ! If this is not the desired behavior, you need to use the `.copy()` method to make a copy of a list (or dictionary).

2 Dictionaries 2

Making copies of a dictionary is the same as with lists. Using the = operator just creates a new reference, but `.copy()` creates a new object.
Last week, we learned about dictionary methods `.items()`, `.keys()`, and `.values()`, and how to use them in a for loop structure. You can use these to create a list of the dictionary keys and values if needed (and for example, assign them to a new variable and use indexing or any of the list methods we learned). Here, `list()` is a constructor that returns a list of any iterable object.

```
languages = {"FORTRAN":1957, "LISP":1958, "COBOL":1959}
print("keys of languages list:", list(languages.keys()))
values = list(languages.values())
print("Second element of values:", values[1])
```

The `.get()` dictionary method returns the value associated with the key passed as an argument to the method. It is very similar to just using the square bracket notation we learned last class. The only difference is that if the key accessed is missing (does not exist in the dictionary), with the `.get()` method, `None` will be returned. On the other hand, if using the square bracket notation with a missing key, you will encounter a KeyError.

```
languages = {"FORTRAN":1957, "LISP":1958, "COBOL":1959}
#both will return 1957
print(languages.get('FORTRAN'))
print(languages['FORTRAN'])
#the next line will return None
print(languages.get('ALGOL')) #not in our dictionary!
#print(languages['ALGOL']) #if not commented out, this line would result in an error
```

The operator + cannot be used with dictionaries to merge them. As the name suggests, the `.update()` method updates a dictionary with the elements from another dictionary.

```
languages_1 = {"FORTRAN":1957, "LISP":1958, "COBOL":1959}
languages_2 = {"SNOBOL": 1962, "BASIC":1964}
print("Original dictionary:", languages_1)
languages_1.update(languages_2)
print("Updated dictionary:", languages_1)
```

If the same key exists in the second dictionary, the value of the first dictionary will be updated:

```
languages_1 = {"FORTRAN":1957, "LISP":1958, "COBOL":1959}
languages_2 = {"FORTRAN":"a programming language"}
print("Original dictionary:", languages_1)
languages_1.update(languages_2)
print("Updated dictionary:", languages_1) #FORTRAN is in both languages_1 and languages_2!
```

The `.pop()` method behaves similarly to lists: It takes one argument, the key of the item to be removed. Note that if the popped element is assigned to a new variable or used in any way, it will be the value associated with the removed element.

```
languages = {"FORTRAN":1957, "LISP":1958, "COBOL":1959}
print("Original dictionary:", languages)
#below, it still returns the removed value, but we're not doing anything with it
languages.pop("LISP")
```

```python
print("Modified dictionary:", languages)
#below, we store the removed value in a new variable popped_elem
popped_elem = languages.pop("COBOL")
print("2x modified dictionary:", languages)
print("The popped element:", popped_elem)
```

Bonus: comprehensions

Note: Using comprehensions is *never required* for the course exercises/assignments, but you are welcome to use them when appropriate.

Consider the following code where elements from a list are added to an (initially empty) new list based on a condition (based on if the string element has the ”L” character or not):

```python
languages = ["FORTRAN", "COBOL", "LISP"]
new_list = [] #empty list

for language in languages:
    if "L" in language:
        new_list.append(language)

print("the new list:", new_list)
```

The above code can be rewritten in a short, concise way using a *list comprehension* using the syntax `[expression for item in iterable if condition == True]`:

```python
languages = ["FORTRAN", "COBOL", "LISP"]

new_list = [language for language in languages if "L" in language]
print("the new list:", new_list)
```

The most basic syntax of creating *dictionary comprehensions* is `{key: value for variables in iterable}`. Consider the following code:

```python
square_dict = {} #empty dictionary
for num in range(2,20,3):
    print(num)
    square_dict[num] = num*num #adding new keys and values to the dictionary
print(square_dict)
```

The above can be rewritten in the following way:

```python
square_dict = {num:num*num for num in range(2,20,3)}
print(square_dict)
```

Try to understand the outputs of the following codes, and rewrite them without using comprehensions!

```python
even_numbers = [x for x in range(1,50) if x % 2 == 0]
print(even_numbers)

my_dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
my_dict2 = {key:value*3 for key,value in my_dict1.items()}
print(my_dict1)
print(my_dict2)
```