UNIVERSITY OF CALIFORNIA, MERCED

BERKELEY · DAVIS · IRVINE · LOS ANGELES · MERCED · RIVERSIDE · SAN DIEGO · SAN FRANCISCO

SANTA BARBARA · SANTA CRUZ

CSE 15: Discrete Mathematics
Laboratory 4

Spring 2020

Introduction

In this lab we continue to practice our Python programming skills by implementing a collection of functions for set theory. Download the file `set_theory.py` from the folder associated with this lab on CatCourses. It contains definitions, and test cases for several functions related to set theory. None of the functions in the file have been implemented. Your task is simple, implement the functions and upload your completed `set_theory.py` file to the appropriate CatCourses assignment.

Useful Python Stuff

Lists

In this lab, we will be representing sets as a Python list. Python lists are a little different than arrays in languages such as Java, that some of you may be familiar with. In those languages, all array elements have to be the same data type. This is not the case for Python lists. In Python we can have a list that contains not only elements of different data types, but some of the elements can be lists themselves. This matches very well with the notion of sets, that we have been exploring in class. For example, these are all lists:

```
A = [1, 2, 3, 4, 5, 6, 7, 8, 9]      # Straightforward, like an array of integers
B = [1, 'a', True, 1.5, 'banana']    # All elements are of a different data type
C = [1, 2, [], ['a', 'b'], 'seven']  # Some of the elements are themselves lists
```

Iterating Over a List

Iterating over lists in Python is a little different than other languages because of the way the Python for-loop works. Instead of having a control variable initialized to a value, a statement to change the control variable, and a stopping condition based on the control variable, as it is in Java (and other languages), the Python for-loop simply takes a list of values and assigns its control variables to each of the values in the list. Consider this code fragment:

```
campuses = ['Merced', 'Davis', 'Berkeley']
for i in campuses:
    print (i)
```

The control variable `i` will first have a value of "Merced", then "Davis", and then "Berkeley". It is important to note that the variable `i` is not an integer index of the list. That is why, if we want to print the name of each campus, we simply `print (i)`, and not `print (campuses[i])`, as it is more common in other languages.

If you want to use the control variable of the for-loop as an index into the list, then you can do:

```
campuses = ['Merced', 'Davis', 'Berkeley']
for i in range(len(campuses)):
    print (campuses[i])
```

In Python, the `len` function returns the length/size of a list or a string. In the example above, that would evaluate to 3, because we have three items in the `campuses` list. The length is then used as a parameter to the `range` function, which generates a list from 0 up to but not including the value of the parameter. So in the example above, it generates the list `[0, 1, 2]`, which corresponds to the indices of the `campuses` list. The control variable `i` now takes on the values from the list `[0, 1, 2]`, so it can be used as an index of the list.

Checking List Membership

A task that you will need to perform a lot in this lab is to check whether a specific value appears in a list. You can use the `in` operator for this. For example if we have the list `A = [1, 3, 5, 7, 9]`, the Python statement `7 in A` evaluates to `True`, whereas `6 in A` is `False`.

Adding to a List

All the examples up to this point have been of Python lists that are manually initialized (hard-coded). It is of course possible to add contents to any Python list, with the help of some built-in list functions. To add an item to the end of the list, namely appending, we use the `append` function called from a list, with the item to append passed as a parameter. For example, if we have a list `A = [1, 3]`, we can execute the Python statement `A.append(5)`, which will change the value of `A` to be the list `[1, 3, 5]`.

We can also insert an element in a different position, say at position 0, which is the beginning of the list. Carrying on our example, the Python statement `A.insert(0, 42)`, will result in the number 42 being inserted at the beginning of `A`, meaning that the contents of the list `A` are `42, 1, 3, 5`.

You can also merge two lists together with the `+` operator. For example `[1, 3, 5] + [2, 4, 6]` evaluates to `[1, 3, 5, 2, 4, 6]`.

Sublists

In Python, as in many other languages, list items are indexed by an integer starting at 0, which corresponds to the first element of the list. Unlike many other languages however, Python assigns the index of -1 to the last element of the list. For example, if we have a list `A = [1, 2, 3]`, and we wished to get the last element, we could write `A[len(A)-1]`, which is the index 1 less than the length of the list. This is fine, but the same result can be obtained by `A[-1]`. Of course, the second last element of `A` is `A[-2]`, and so forth.

In addition to getting single list elements, Python allows us to take a sublist by specifying a range, instead of a single value. For example, given a list `A = [5, 7, 3]`, the statement `A[0:2]`, evaluates to the list starting at position 0 up to but not including position 2, which results in the list `[5, 7]`. If we omit the lower bound of the range, it defaults to 0. So the statement `A[:2]` is equivalent to `A[0:2]`. Similarly, if we omit the upper bound, it defaults to the length of the list, in other words it will copy to the end. Examples:

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

A[2:4] # evaluates to [3, 4]

A[:3]  # evaluates to [1, 2, 3]

A[2:]  # evaluates to [3, 4, 5]
```

Copying Lists

In Python, as it is with many other languages, to make a copy of a variable, we just assign another variable to the one we want to copy. For example, if we have `x = 7`, and we want to copy `x`, we simply say `y = x`. We now have another variable `y`, which is a copy of `x`. For `y` to be a copy of `x`, it needs to have the same value as `x`, which it does. It also exists independently of `x`, which means that, continuing our example, at this point both `x` and `y` have the value of 7. If we now decide to change the value of one of them, you would expect that the the other one is not modified as well, right? For example, if we say `y = 9`, we will now have `y` with a value of 9, and `x` with a value of 7. That is how it should be when we make a copy.

Unfortunately, lists in Python do not behave in this way. So if we have a list `A = [1, 2, 3]`, and we make a copy with `B = A`, we now have `A` and `B`, which are both `[1, 2, 3]`. So far so good, but if we modify `B`, say with `B.append(7)`, we would now expect `B` to have a value of `[1, 2, 3, 7]`, which it does, but upon inspection we find out that `A` is also `[1, 2, 3, 7]`. How can that be if we only stated that `B` should be appended? The answer is that when we made `B` to be a copy of `A`, Python only made what is known as a *shallow copy*. What this means is that `B` is not a different list which has the same values as `A`, it is actually just a *pointer* to `A`, so `A` and `B` are actually the same thing, which means that if one changes, the other one changes too.

To create a *deep copy* of `A`, we use the fact that when taking a sublist, the result is an actual deep copy of the original. In our running example, to define the list `C` as a deep copy of `A`, we need to say `C = A[:]`. If you experiment by changing something in `C`, you will find that those changes are not reflected in `A`. Likewise, changes in `A` will not be reflected in `C`, because it is a deep copy, which exists independently of the original.

Tuples

Tuples are another container class in Python. They are specified in parentheses, for example `j = (7, 3)`. Tuples can also be indexed, like lists. By saying `j[0]` and `j[1]`, we can get to the 7 and the 3 respectively. The difference between tuples and lists is that tuples can not be modified, they are immutable.

Useful Programming Stuff

Checking a Condition on List Elements

Many of the exercises in this lab will involve checking whether a certain condition is satisfied for every list element. There is a useful construct, sometimes referred to as a *flag*, which keeps track of the condition as we are going through the loop. To make use of it, we have been given a list, and we assume the condition is true for every element. We set up the code like this:

```
theList = [...]

flag = True           # Let's assume the condition we are testing is true
for i in theList:
    if condition does not hold for i:
        # We found an element that does not meet the condition, we are done
        flag = False
        break

# When we are done with the for-loop, the flag will either be True of False
# corresponding to whether the condition is true for all elements of the list.

# Since we started with flag true, and the only way to make it false is to
# find a list element for which the condition is not true, this means that if
# flag is false at this point of the code, we must have found a list element that
# does not meet the condition, so the condition can't hold for every element.

# If on the other hand, flag is still true, this means that we never made it false
# in the loop, meaning the condition must be true for all elements of the list.
```

Building up a List

Sometimes the solution to a problem is to go through a given list (possibly more than one list), and select only those elements for which a certain condition holds, and make that the result. We set up the code in the following way:

```
inputList = [...]

result = []  # At the start result is empty

for i in inputList:
    if condition holds for i:
        result.append(i)

# At this point result is made up of only those input elements that met the condition.
```