LISTS

CS 3080: Python Programming

University of Colorado
Colorado Springs

All my cats

- Normally is tempting to create many individual variables to store a group of similar values

```
catName1 = 'Zophie’
catName2 = 'Pooka’
catName3 = 'Simon’
catName4 = 'Lady Macbeth’
catName5 = 'Fat-tail’
catName6 = 'Miss Cleo'
```

All my cats

```
print('Enter the name of cat 1:’)
catName1 = input()
print('Enter the name of cat 2:’)
catName2 = input()
print('Enter the name of cat 3:’)
catName3 = input()
print('Enter the name of cat 4:’)
catName4 = input()
print('Enter the name of cat 5:’)
catName5 = input()
print('The cat names are:’)
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' +
catName4 + ' ' +catName5)
```

What if the number of cats changes?

All my cats

- Instead of using multiple, repetitive variables, you can use a single variable that contains a list value

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
          ' (Or enter nothing to stop.):’)
    name = input()
    if name == ’’:
        break
    catNames = catNames + [name]  # list concatenation

print('The cat names are:’)
for name in catNames:
    print('  ' + name)
```

List definition: a number of connected items

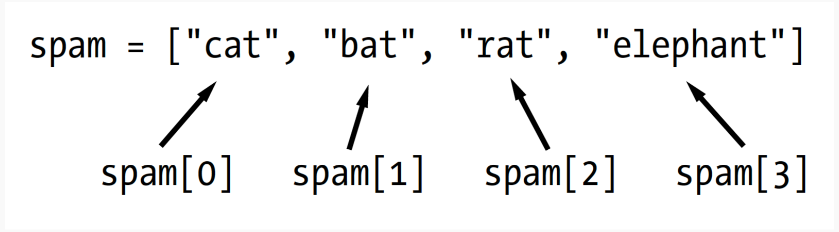
Lists and tuples

- Lists and tuples can contain multiple values, which makes it easier to write programs that handle large amounts of data.
- And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

List

- A **list** is a value that contains multiple values in an ordered sequence
- The term **list value** refers to the list itself, not the values inside the list value.
  - *['cat', 'bat', 'rat', 'elephant’]*
- Values inside the list are called **items**.
- The value [] is an empty list that contains no values, similar to '', the empty string

Getting Individual Values in a List with Indexes



```
spam = ["cat", "bat", "rat", "elephant"]

    spam[0]   spam[1]   spam[2]   spam[3]
```

- Python will give you an IndexError error message if you use an index that exceeds the number of values in your list value.
  - Try spam[1000]

- Indexes can be only integer values, not floats.
  - Try spam[1.0]

- Lists can also contain other list values.

Negative Indexes

- While indexes start at 0 and go up, you can also use **negative integers** for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on.
  - *Try spam[-1], spam[-2]*

Sublists with Slices

- A **slice** can get several values from a list, in the form of a new list.
  - *Slice -> 1:4*
  - *Slice in a list -> spam[1:4]*
- The first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but **will not include, the value at the second index**.
- Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.
  - *spam[:4]*
  - *spam[2:]*
  - *spam[2:None]*

Extended slices

- a[n:m:k] returns every kth element from n to m

```
L = [0, 1, 2, 3, 4, 5, 6]
L[0:4:2]                # [0, 2]
L[::2]                  # [0, 2, 4, 6]
L[::-1]                 # [6, 5, 4, 3, 2, 1, 0]
```

len()

- The len() function will return the number of values that are in a list value passed to it.
  - *Try len(spam)*

Change list values

- You can use an index of a list to change the value at that index
  - *spam[1] = 'aardvark'*

Change list values

```
L = [0, 1, 2, 3, 4, 5, 6]
L[2] = 10                     # [0, 1, 10, 3, 4, 5, 6]
L[-2] = 10                    # [0, 1, 2, 3, 4, 10, 6]
L[2:4] = [10, 10]             # [0, 1, 10, 10, 4, 5, 6]
L[::3] = [10, 10, 10]         # [10, 1, 2, 10, 4, 5, 10]
```

List Concatenation and List Replication

- + operator >> combine two lists to create a new list value
  - *[1, 2, 3] + ['A', 'B', 'C']*
- * operator >> used with a list and an integer value to replicate the list
  - *['X', 'Y', 'Z'] * 3*

del statement

- **del** statement will delete values at an index in a list
- All of the values in the list after the deleted value will be moved up one index.
  - *del spam[2]*

Using for Loops with Lists

```
for i in range(4):
    print(i)
```

- Output?

Using for Loops with Lists

```
for i in range(4):
    print(i)
```

=

```
for i in [0, 1, 2, 3]:
    print(i)
```

- Output?

- Technically, a for loop repeats the code block once for each value in a list or list-like value.
- This is because the return value from range(4) is a list-like value that Python considers similar to [0, 1, 2, 3].
- A common Python technique is to use *range(len(someList))* with a for loop to iterate over the indexes of a list

The in and not in Operators

- You can determine whether a value is or isn’t in a list with the **in** and **not in** operators.
- Like other operators, in and not in are used in expressions and connect two values:
  - *a value to look for in a list*
  - *the list where it may be found.*
- These expressions will evaluate to a Boolean value.
  - *'howdy' in ['hello', 'hi', 'howdy', 'heyas']*

The Multiple Assignment Trick

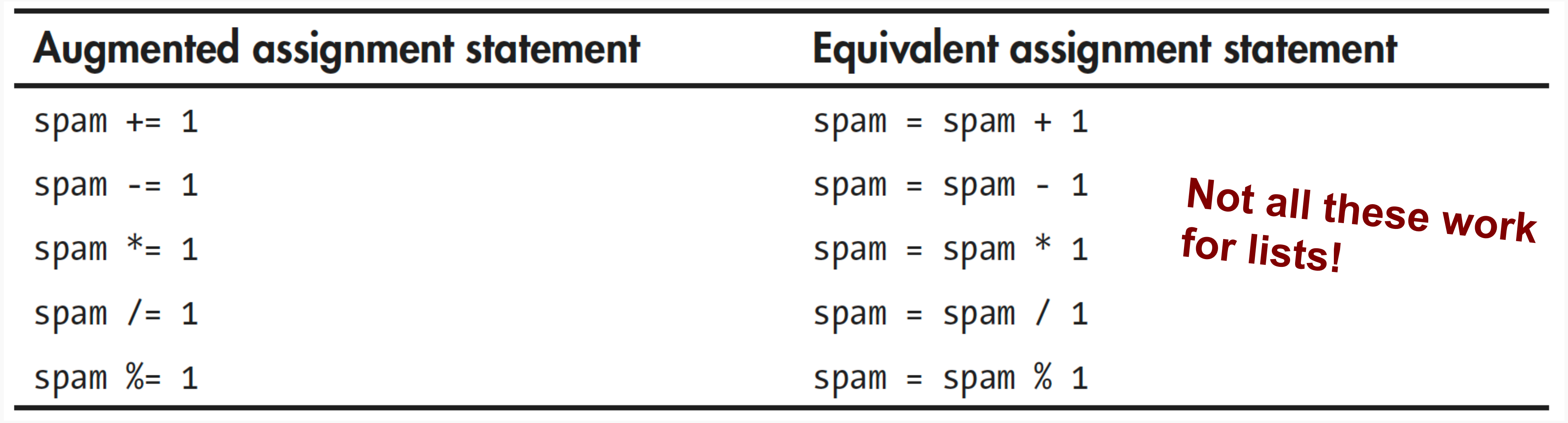
- The **multiple assignment trick** is a shortcut that lets you assign multiple variables with the values in a list in one line of code.

```
cat = ['fat', 'black', 'loud']
size = cat[0]
color = cat[1]
disposition = cat[2]
```

Same as:

```
cat = ['fat', 'black', 'loud']
size, color, disposition = cat
```

Augmented Assignment Operators



Augmented assignment statement	Equivalent assignment statement
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

Not all these work for lists!

The += operator can also do string and list concatenation, and the *= operator can do string and list replication.

```
spam = [‘Bob’]
spam *= 3     # spam = [‘Bob’, ‘Bob’, ‘Bob’]
```

Method

- A **method** is the same thing as a function, except it is “called on” a value
- Each data type has its own set of methods.

List methods – .index(‘value’)

- **index()** method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn’t in the list, then Python produces a ValueError error.
- When there are duplicates of the value in the list, the index of its first appearance is returned.

List methods – .append(‘va’) and .insert(1, ‘va’)

- **append()** method call adds the argument to the end of the list
  - *spam.append(‘moose’)*
- **insert()** method can insert a value at any index in the list
  - *spam.insert(1,‘chicken’)*
- The return value of append() and insert() is **None**

List methods – .remove(‘value’)

- **remove()** method is passed the value to be removed from the list it is called on.
  - *spam.remove(‘cat’)*
- Attempting to delete a value that does not exist in the list will result in a ValueError error.
- If the value appears multiple times in the list, only the first instance of the value will be removed.

List methods – .sort()

- Lists of number values or lists of strings can be sorted with the **sort()** method.
  - *spam.sort()*
- You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order.
  - *spam.sort(reverse=True)*
- You **cannot** sort lists that have different data types values in them
- Uses “ASCIIbetical order”
  - *If you need to sort the values in regular alphabetical order use:*
  - *spam.sort(key=str.lower)*

Mutable and immutable data types

- **Mutable** data type >> It can have values added, removed, or changed.
  - *list*
- **Immutable** data type >> It cannot be changed.
  - *string*

Tuple Data Type

- The **tuple** data type is almost identical to the list data type, except for:
  - *are typed with parentheses, eggs = ('hello', 42, 0.5)*
  - *are* ***immutable***

- It’s fine to have a trailing comma after the last item in a list or tuple. Is used when you have only one value to let Python know that it is a tuple value
  - *type(('hello’,)) → tuple*
  - *type(('hello’)) → str*

List or tuple?

- You can use both, but:
  - *tuples are immutable and their contents don’t change. So Python can implement some optimizations that make code using tuples slightly faster than code using lists.*

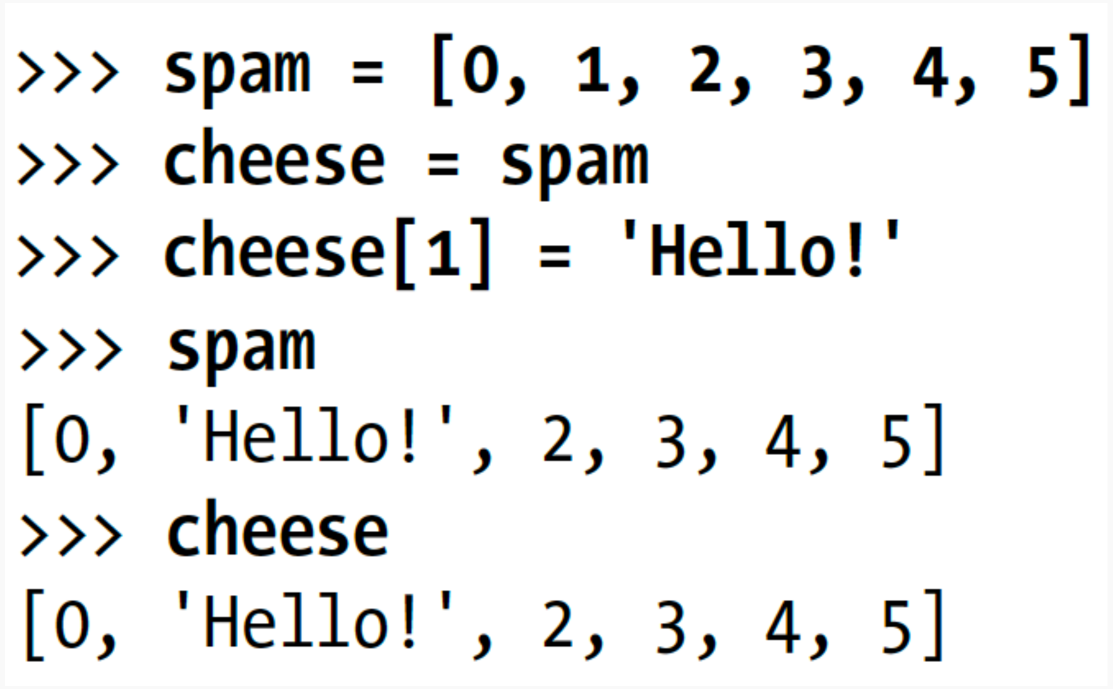
- So **if your content doesn’t change, use a tuple.**

- Converting between them:

```
tuple(['cat', 'dog', 5])
list(('cat', 'dog', 5))
```

List references

- When you assign a list to a variable, you are actually assigning a list reference to the variable.
- A reference is a value that points to some bit of data, and a list reference is a value that points to a list.



```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

List references

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

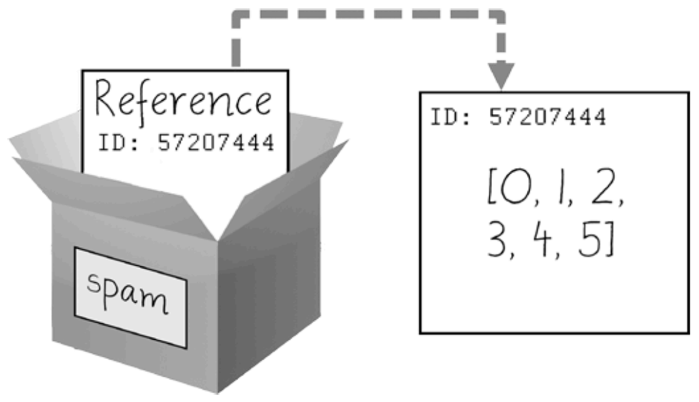


Figure 4-4: spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

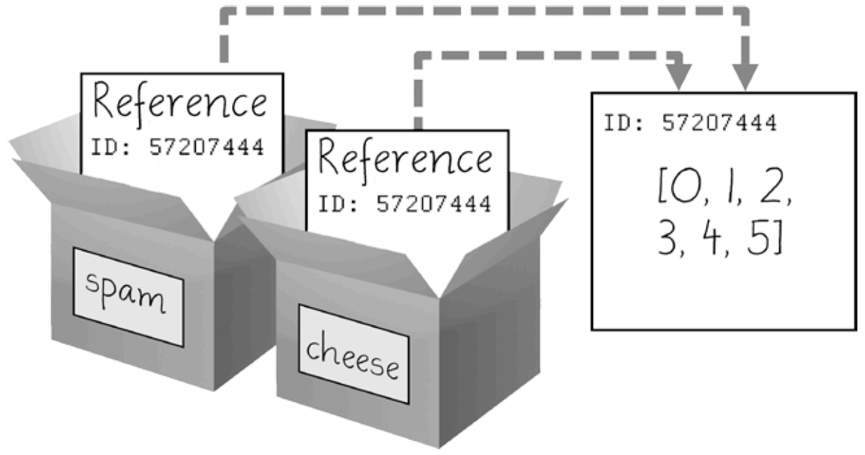


Figure 4-5: spam = cheese copies the reference, not the list.

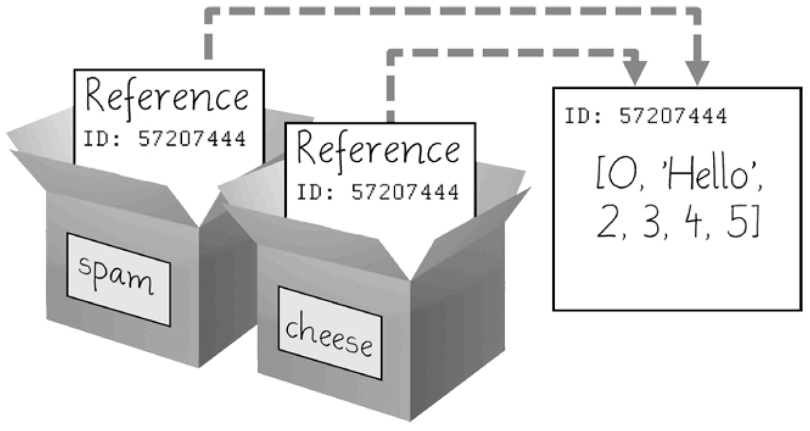


Figure 4-6: cheese[1] = 'Hello!' modifies the list that both variables refer to.

Python references

- Python uses **references** whenever variables must store values of **mutable data types**, such as lists or dictionaries.
- For values of **immutable data types** such as strings or tuples, Python variables will store the **value itself**.

Passing references in function arguments

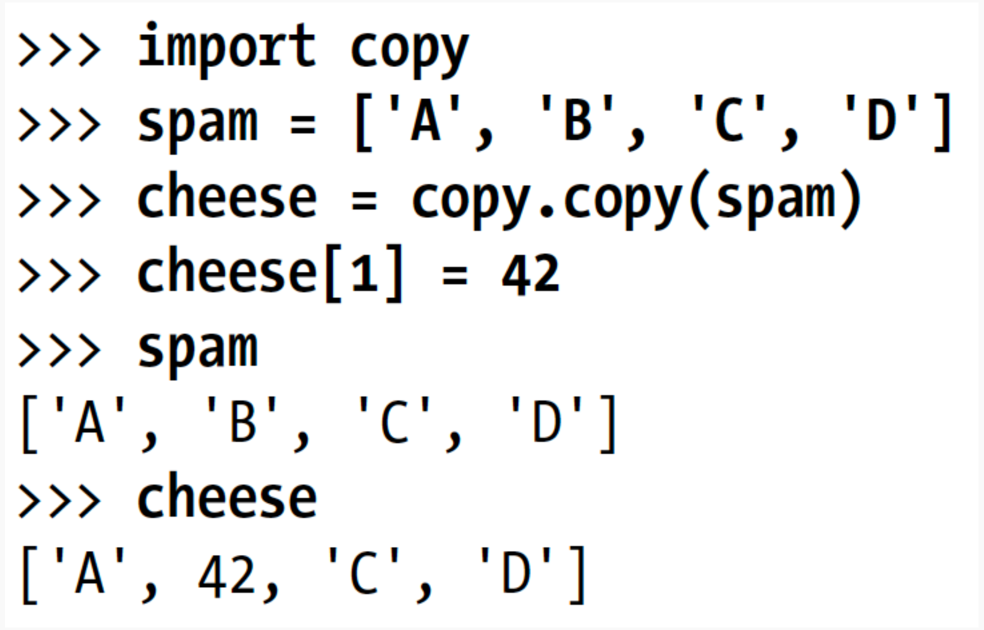
- When a function is called, the values of the arguments are copied to the parameter variables.
- **For mutable data types, a copy of the reference is used for the function arguments**.
  - ***Keep this behavior in mind****: Forgetting that Python handles mutable data types like lists and dictionaries variables this way can lead to confusing bugs.*

Very important to understand this!

The copy Module’s copy() and deepcopy() Functions

- You may not want the changes in the original list or dictionary value.
- *module copy*
  - ***copy.copy(),*** *can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference.*
  - *If the list you need to copy contains lists, the* ***copy.deepcopy()*** *function will copy these inner lists as well.*

The copy Module’s copy() and deepcopy() Functions



```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

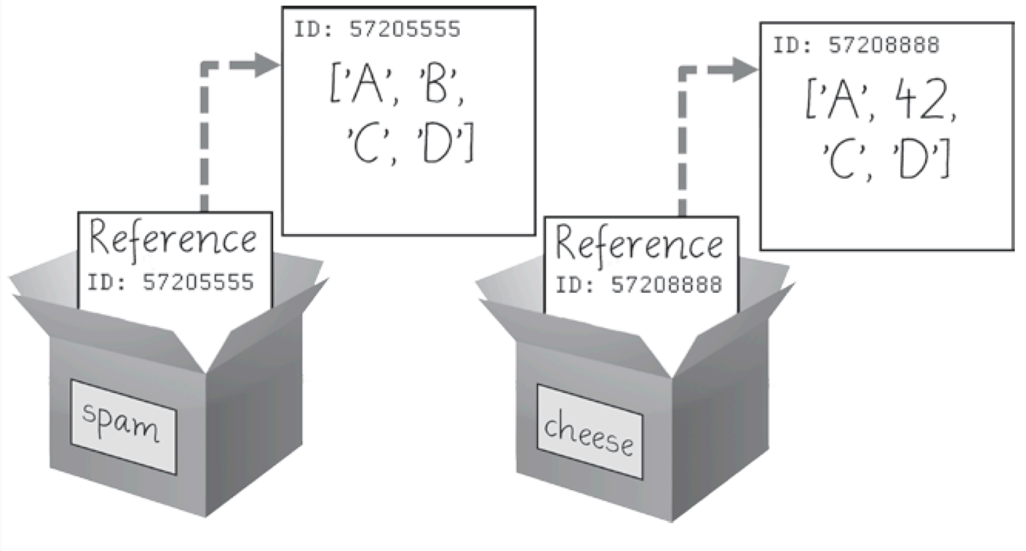


Figure 4-7: cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

List comprehensions

- List comprehension is an easy way to define and create list in Python.

# old way:

```
my_list = []
for x in range(10):
     my_list.append(x * 2)

print(my_list)    # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

List comprehensions

- List comprehension is an easy way to define and create list in Python.

```
my_list = [x * 2 for x in range(10)]

print(my_list)    # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Syntax: [expression for item in list]

```
# Although this is the same as list(range(0, 19, 2))
```

List comprehensions

- You can add conditions to list comprehensions

```
comp_list = [x ** 2 for x in range(7) if x % 2 == 0]
print(comp_list)  # [0, 4, 16, 36]
```

List comprehensions

- You create lists of lists too.

This is like nested for loops

```
nums = [1, 2, 3, 4, 5]
letters = ['A', 'B', 'C', 'D', 'E']
nums_letters = [[n, l] for n in nums for l in letters]
print(nums_letters)
# [[1, 'A'], [1, 'B'], [1, 'C'], [1, 'D'], [1, 'E'], [2, 'A’],
# [2, 'B'], [2, 'C'], [2, 'D'], [2, 'E'], [3, 'A’], [3, 'B’],
# ...[5, 'D'], [5, 'E']]
```