Lists

WE'LL COVER THE FOLLOWING

Lists are Python's workhorse datatype. When I say "list," you might be thinking "array whose size I have to declare in advance, that can only contain items of the same type, &c." Don't think that. Lists are much cooler than that.

> *A list in Python is like an array in Perl 5. In Perl 5, variables that store arrays always start with the @ character; in Python, variables can be named anything, and Python keeps track of the datatype internally.*

> *A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the **ArrayList class**, which can hold arbitrary objects and can expand dynamically as new items are added.*

Creating A List

Creating a list is easy: use square brackets to wrap a comma-separated list of values.

```
a_list = ['a', 'b', 'mpilgrim', 'z', 'example']          #①
print (a_list)
#['a', 'b', 'mpilgrim', 'z', 'example']

print (a_list[0] )                                       #②
#a

print (a_list[4] )                                       #③
#example

print (a_list[-1])                                       #④
#example

print (a_list[-3])                                       #⑤
#mpilgrim
```

① First, you define a list of five items. Note that they retain their original order. This is not an accident. A list is an ordered set of items.

② A list can be used like a zero-based array. The first item of any non-empty list is always `a_list[0]`.

③ The last item of this five-item list is `a_list[4]`, because lists are always zero-based.

④ A negative index accesses items from the end of the list counting backwards. The last item of any non-empty list is always `a_list[-1]`.

⑤ If the negative index is confusing to you, think of it this way: `a_list[-n] == a_list[len(a_list) - n]`. So in this list, `a_list[-3] == a_list[5 - 3] == a_list[2]`.

Slicing A List

a_list[0] is the first item of **a_list**.

Once you've defined a list, you can get any part of it as a new list. This is called *slicing* the list.

```
a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
```

```
print (a_list)
#['a', 'b', 'mpilgrim', 'z', 'example']

print (a_list[1:3] )           #①
#['b', 'mpilgrim']

print (a_list[1:-1])           #②
#['b', 'mpilgrim', 'z']

print (a_list[0:3] )           #③
#['a', 'b', 'mpilgrim']

print (a_list[:3])             #④
#['a', 'b', 'mpilgrim']

print (a_list[3:] )            #⑤
#['z', 'example']

print (a_list[:] )             #⑥
#['a', 'b', 'mpilgrim', 'z', 'example']
```

① You can get a part of a list, called a “slice”, by specifying two indices. The return value is a new list containing all the items of the list, in order, starting with the first slice index (in this case `a_list[1]`), up to but not including the second slice index (in this case `a_list[3]`).

② Slicing works if one or both of the slice indices is negative. If it helps, you can think of it this way: reading the list from left to right, the first slice index specifies the first item you want, and the second slice index specifies the first item you don’t want. The return value is everything in between.

③ Lists are zero-based, so `a_list[0:3]` returns the first three items of the list, starting at `a_list[0]`, up to but not including `a_list[3]`.

④ If the left slice index is 0, you can leave it out, and 0 is implied. So `a_list[:3]` is the same as `a_list[0:3]`, because the starting 0 is implied.

⑤ Similarly, if the right slice index is the length of the list, you can leave it out. So `a_list[3:]` is the same as `a_list[3:5]`, because this list has five items. There is a pleasing symmetry here. In this five-item list, `a_list[:3]` returns the first 3 items, and `a_list[3:]` returns the last two items. In fact, `a_list[:n]` will always return the first n items, and `a_list[n:]` will return the rest, regardless of the length of the list.

⑥ If both slice indices are left out, all items of the list are included. But this is

⑤ If both slice indices are left out, all items of the list are included. But this is not the same as the original `a_list` variable. It is a new list that happens to have all the same items. `a_list[:]` is shorthand for making a complete copy of a list.

Adding Items To A List

There are four ways to add items to a list.

```
a_list = ['a']
a_list = a_list + [2.0, 3]          #①
print (a_list )                     #②
#['a', 2.0, 3]

a_list.append(True)                 #③
print (a_list)
#['a', 2.0, 3, True]

a_list.extend(['four', '/'])        #④
print (a_list)
#['a', 2.0, 3, True, 'four', 'Ω']

a_list.insert(0, '/')               #⑤
print (a_list)
#['Ω', 'a', 2.0, 3, True, 'four', 'Ω']
```

① The `+` operator concatenates lists to create a new list. A list can contain any number of items; there is no size limit (other than available memory). However, if memory is a concern, you should be aware that list concatenation creates a second list in memory. In this case, that new list is immediately assigned to the existing variable `a_list`. So this line of code is really a two-step process — concatenation then assignment — which can (temporarily) consume a lot of memory when you're dealing with large lists.

② A list can contain items of any datatype, and the items in a single list don't all need to be the same type. Here we have a list containing a string, a floating point number, and an integer.

③ The `append()` method adds a single item to the end of the list. (Now we have *four* different datatypes in the list!)

④ Lists are implemented as classes. "Creating" a list is really instantiating a class. As such, a list has methods that operate on it. The `extend()` method

takes one argument, a list, and appends each of the items of the argument to the original list.

⑤ The `insert()` method inserts a single item into a list. The first argument is the index of the first item in the list that will get bumped out of position. List items do not need to be unique; for example, there are now two separate items with the value `'Ω'`: the first item, `a_list[0]`, and the last item, `a_list[6]`.

> **a_list.insert(0, value)** is like the **unshift()** function in Perl. It adds an item to the beginning of the list, and all the other items have their positional index bumped up to make room.*

Let's look closer at the difference between `append()` and `extend()`.

```
a_list = ['a', 'b', 'c']
a_list.extend(['d', 'e', 'f'])                #①
print (a_list)
#['a', 'b', 'c', 'd', 'e', 'f']

print (len(a_list))                           #②
#6

print (a_list[-1])
#'f'

a_list.append(['g', 'h', 'i'])                #③
print (a_list)
#['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]

print (len(a_list))                           #④
#7

print (a_list[-1])
#['g', 'h', 'i']
```

① The `extend()` method takes a single argument, which is always a list, and adds each of the items of that list to `a_list`.

② If you start with a list of three items and extend it with a list of another three items, you end up with a list of six items.

three items, you end up with a list of six items.

③ On the other hand, the `append()` method takes a single argument, which can be any datatype. Here, you're calling the `append()` method with a list of three items.

④ If you start with a list of six items and append a list onto it, you end up with... a list of seven items. Why seven? Because the last item (which you just appended) *is itself a list*. Lists can contain any type of data, including other lists. That may be what you want, or it may not. But it's what you asked for, and it's what you got.

Searching For Values In A List

```
a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
print (a_list.count('new'))       #①
#2

print ('new' in a_list)           #②
#True

print ('c' in a_list)
#False

print (a_list.index('mpilgrim'))  #③
#3

print (a_list.index('new'))       #④
#2

print (a_list.index('c') )        #⑤
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 17, in <module>
# print (a_list.index('c') ) #\u2464
#ValueError: 'c' is not in list
```

① As you might expect, the `count()` method returns the number of occurrences of a specific value in a list.

② If all you want to know is whether a value is in the list or not, the `in` operator is slightly faster than using the `count()` method. The `in` operator always returns `True` or `False`; it will not tell you how many times the value appears in the list.

③ Neither the `in` operator nor the `count()` method will tell you *where* in the list a value appears. If you need to know where in the list a value is, call the `index()` method. By default it will search the entire list, although you can specify an optional second argument of the (0-based) index to start from, and even an optional third argument of the (0-based) index to stop searching.

④ The `index()` method finds the *first* occurrence of a value in the list. In this case, `'new'` occurs twice in the list, in `a_list[2]` and `a_list[4]`, but the `index()` method will return only the index of the first occurrence.

⑤ As you might *not* expect, if the value is not found in the list, the `index()` method will raise an exception.

Wait, what? That's right: the `index()` method raises an exception if it doesn't find the value in the list. This is notably different from most languages, which will return some invalid index (like -1). While this may seem annoying at first, I think you will come to appreciate it. It means your program will crash at the source of the problem instead of failing strangely and silently later. Remember, -1 is a valid list index. If the `index()` method returned -1, that could lead to some not-so-fun debugging sessions!

Removing Items From A List

Lists never have gaps.

Lists can expand and contract automatically. You've seen the expansion part. There are several different ways to remove items from a list as well.

```
a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
print (a_list[1])
#'b'

del a_list[1]                  #①
print (a_list)
#['a', 'new', 'mpilgrim', 'new']

print (a_list[1])              #②
#new
```

① You can use the `del` statement to delete a specific item from a list.

② Accessing index 1 after deleting index 1 does *not* result in an error. All items after the deleted item shift their positional index to "fill the gap" created by deleting the item.

Don't know the positional index? Not a problem; you can remove items by value instead.

```
a_list = ['a', 'new', 'mpilgrim', 'new']
a_list.remove('new')                        #①
print (a_list)
#['a', 'mpilgrim', 'new']

a_list.remove('new')                        #②
print (a_list)
#['a', 'mpilgrim']

print (a_list.remove('new'))
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 10, in <module>
# print (a_list.remove('new'))
#ValueError: list.remove(x): x not in list
```

① You can also remove an item from a list with the `remove()` method. The `remove()` method takes a *value* and removes the first occurrence of that value from the list. Again, all items after the deleted item will have their positional indices bumped down to "fill the gap." Lists never have gaps.

② You can call the `remove()` method as often as you like, but it will raise an exception if you try to remove a value that isn't in the list.

Removing Items From A List: Bonus Round

Another interesting list method is `pop()`. The `pop()` method is yet another way to remove items from a list, but with a twist.

```
a_list = ['a', 'b', 'new', 'mpilgrim']
print (a_list.pop())   #①
#mpilgrim

print (a_list)
#['a', 'b', 'new']
```

```
print (a_list.pop(1) ) #②
#b


print (a_list)
#['a', 'new']

print (a_list.pop())
#new

print (a_list.pop())
#a

print (a_list.pop())   #③
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 20, in <module>
# print (a_list.pop()) #\u2462
#IndexError: pop from empty list
```

① When called without arguments, the `pop()` list method removes the last item in the list *and returns the value it removed*.

② You can pop arbitrary items from a list. Just pass a positional index to the `pop()` method. It will remove that item, shift all the items after it to "fill the gap," and return the value it removed.

③ Calling `pop()` on an empty list raises an exception.

> *Calling the **pop()** list method without an argument is like the **pop()** function in Perl. It removes the last item from the list and returns the value of the removed item. Perl has another function, **shift()**, which removes the first item and returns its value; in Python, this is equivalent to **a_list.pop(0)**.*

Lists In A Boolean Context

Empty lists are false; all other lists are true.

You can also use a list in a boolean context, such as an if statement.

```
def is_it_true(anything):
   if anything:
     print("yes, it's true")
   else:
```

```
  else:
    print("no, it's false")

print (is_it_true([]) )            #①
#no, it's false
#None

print (is_it_true(['a']))          #②
#yes, it's true
#None

print (is_it_true([False]) )       #③
#yes, it's true
#None
```

① In a boolean context, an empty list is false.

② Any list with at least one item is true.

③ Any list with at least one item is true. The value of the items is irrelevant.