Teach Python 2: Lists and Strings

List Basics

Lists are dynamic, array-like structures. They are the closest thing Python has to arrays.

List literals use square brackets (e.g. `[1,2,3]`, `['one', 'two', 3]`, `[]`).

List elements are accessed using square brackets (e.g. `a = b[0]` or `b[0] = a`).

List elements can be of any type, including other lists. The elements of a list do not have to be the same type.

Lists can be concatenated with + and repeated with `*`, just like strings (see handout 1).

Python contains a number of built-in functions that operate on lists.

Lists are also objects with methods (see Appendix A).

String Basics

Strings are immutable lists of characters.

String literals use single or double quotes (e.g. `"Hello"`, `'Hi'`, `""`).

String elements are accessed using square brackets (e.g. `t = s[0]` but not `s[0] = t`, because strings are immutable).

Python does not have a character type, so if `s` is `"Hello"`, then `s[0]` is the string `"H"`.

Strings can be concatenated with + and repeated with * (see handout 1).

Python contains a number of built-in functions that operate on strings.

Strings are also objects with methods (see Appendix A).

Negative Indices

Python supports negative indices. A negative index is subtracted from the length of the list or string to yield a positive index.

Legal positive indices range from 0 to length-1. Legal negative indices range from -1 to -length. Out of range indices cause an `IndexError` exception to be raised.

Example: `a = [32, 5, 1, 43]`

Index	**-5**	**-4**	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**
Value	Error	32	5	1	43	32	5	1	43	Error

Slices

Python contains a powerful operator known as the “slice” operator. When indexing into a list or string, you can use the colon to specify a range of indices, like this `[start:end]`. The range will include index `start`, but not index `end`. The value of the expression returned will be a copy of a (possibly empty) “slice” of the list you specified.

Examples: `a = [32, 5, 1, 43]` `b="python”`

Expression	**a[1]**	**a[1:3]**	**a[1:2]**	**a[1:1]**	**b[1]**	**b[1:3]**	**b[1:2]**	**b[1:1]**
Value	5	[5,1]	[5]	[]	'y'	'yt'	'y'	''

You can leave out the start or end value (it defaults to the start or end of the entire list or string), and you can use negative indices in a slice.

Examples: `a = [32, 5, 1, 43]` `b="python”`

Expression	**a[:2]**	**a[2:]**	**a[:-1]**	**b[:2]**	**b[2:]**	**b[:-1]**
Value	[32, 5]	[1, 43]	[32,5,1]	'py'	'thon'	'pytho'

Note that `a[:]` is a quick way to make a copy of `a`.

With lists (but not strings) you can also assign to a slice.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a[2:4] = [0, 0, 0]    # replaces the slice with the given list
```

Inserting and Appending

For lists, you can insert and append using slices.

```
a[0:0] = ['put me', 'first!!!']  # inserts at the beginning

ln = len(a)                      # gets the length of the list
a[ln:ln] = ['put me', 'last!!!'] # appends to the end

a[5:5] = ['put me', 'inside']    # inserts before element 5
```

For lists and strings, you can use the + operator to create a new list or string with new elements inserted. The examples below mirror the examples above, first for lists and then for strings.

```
a = ['put me', 'first!!!'] + a
a = a + ['put me', 'last!!!']
a = a[:5] + ['put me', 'inside'] + a[5:]

a = 'put me first!!!' + a
a = a + 'put me last!!!'
a = a[:5] + 'put me inside' + a[5:]
```

Deleting

For lists, you can remove an index or a slice using the `del` operator:

```
del a[10]  # removes element 10 from a
del a[5:8] # removes the slice 5:8
```

For lists and strings, you can also use the + operator to make a copy of the list or string with some elements removed.

```
a = a[:5] + a[8:]
```

Testing Membership

Python has a Boolean operator called `in` for testing membership of elements in lists or substrings within strings. Expressions like the ones below can be embedded in `if` statements and `while` loops.

`x in y`	Returns `True` if `x` is an element/substring of `y`
`x not in y`	Returns `True` if `x` is not an element/substring of `y`

Some Useful Built-In Functions

The functions below will work whether `a` is a string or a list.

`len(a)`	Returns the length of `a`
`min(a)`	Returns the minimum element in `a`
`max(a)`	Returns the maximum element in `a`
`sorted(a)`	Returns a new sorted list or string with the same contents as `a`
`sorted(a,reverse=True)`	As above, but in descending order

The function below will only work if `a` is a list.

`sum(a)`	Returns the sum of the elements in `a`

The `for` Loop

Lists and strings can be iterated over using a `for` loop. If the variable `y` holds a list, the loop below will print every element of the list. If `y` contains a string, it will print every character. It works by placing each list item (or each character as a singleton string) into the variable `x` (one at a time) and the executing the loop body for that value of `x`.

```
for x in y:
    print(x)
```

The `range` Function[1]

If you want to use the `for` loop as a counting loop, you can create a `range` object. A `range` object is a sequence of integers. You can specify just the end point of the range (it will start at 0 and count by 1's), or both the start and end, or the start, end and step or "count by" value. Here are some examples:

`range(10)`	= [0,1,2,3,4,5,6,7,8,9]	← [0, End)
`range(1,11)`	= [1,2,3,4,5,6,7,8,9,10]	← [Start, End)
`range(1,11,2)`	= [1, 3, 5, 7, 9]	← [Start, End) but stepping by 2's.

Note the range does not include the end value.

Here's an example of a loop that counts from 0 to 9:

```
for i in range(10):
    print(i)
```

And here's one that counts from 5 to 10000 by 5's:

```
for i in range(5, 10001, 5):
    print(i)
```

Use range in conjunction with the `len` function to process a list when you need to make changes to it:

```
for i in range(len(x)):
    x[i] = x[i] ** 2
```

The `reversed` Function

The built-in function `reversed` will reverse a range, list or string. You can use it to visit the elements of a string or list in reverse order, either by reversing the list or string itself:

```
for e in reversed(x)
    print(e)
```

Or by reversing the indices of the list or string:

```
for i in reversed(range(len(x)))
    print(x[i])
```

The `enumerate` Function

The built-in function `enumerate` can be used to retrieve both the element and its index.

```
for i, e in enumerate(x):
    print('Element', i, 'is:' ,e)
    x[i] = e * 2
```

[1] The `range`, `reversed` and `enumerate` "functions" are actually constructors for creating `range`, `reversed` and `enumerate` objects.

Appendix A: Methods

Both strings and lists are objects with many useful methods.

The methods below will work whether `a` is a string or a list. Parameter `e` should be a list item or a string as appropriate.

`a.index(e)`	Returns the index of the first element equal to `e`
	(Raises a `ValueError` exception if `e` is not in the list or string)
`a.count(e)`	Returns a count of the number of times `e` appears in `a`

The methods below will work only if `a` is a list.

`a.append(e)`	Adds element `e` to the end of `a`
`a.extend(b)`	Adds all elements from list `b` to the end of `a`
`a.insert(i, e)`	Inserts `e` into `a` at index `i`
`a.remove(e)`	Deletes the first element that is equal to `e` (using ==)
`a.sort()`	Sorts the list in place
`a.sort(reverse=True)`	As above, but in descending order

The methods below will work only if `a` is a string. None of these modify the original string, but some do return a new version of the string with changes made.

`a.find(s)`	Just like the `index` method but returns -1 if `s` not found.
`a.lower()`	Convert to lowercase
`a.upper()`	Convert to uppercase
`a.replace(s, t)`	Replace all occurrences of substring `s` with string `t`
`a.strip()`	Remove leading and trailing whitespace
`a.split()`	Splits the string on whitespace and returns a list of words

For complete documentation and lost more methods, use `help(str)` or `help(list)`.

Appendix B: Exceptions

There are two exceptions that you might see in code using lists and strings:

```
a = [1,2,3]
b = int(input())
a[b] = 10               # this might cause an IndexError exception

def spam(a):
     print(a[0])        # this might cause a TypeError if a is
                        # not a list. It could also cause an
                        # IndexError if a is the empty list.
```

You can catch exceptions or you can avoid them.

To avoid an `IndexError`, make sure that for a list of length L, the index i satisfies $-L \leq i < L$.

To avoid a `TypeError` use the Boolean function `isinstance` to make sure a variable holds a list:

```
def eggs(a):
     if isinstance(a, list):
          print(a[0])
     else:
          print('ERROR: the parameter was not a list')
```

Tip: don't call your list variables 'list'. This is a built-in name for the list type and if you change it, the above code won't work any more.