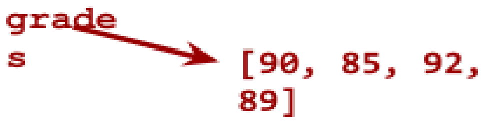
DS2000
Fall 2021
Handout: Lists

A list is an ordered collection of values that are grouped together under a single variable name. We know that programming languages are meant to be intuitive, so think of a Python list like any other kind of a list: to-do list, grocery list, enemies list, etc. When you create a list, it is one variable with multiple items; every item has a *value* as well as a *position* within the list.

Lists are identified in Python with square brackets. Here is how I would initialize a list of 4 integers:

```
grades = [90, 85, 92, 89]
```

In memory, **`grades`** is a label, just like it would be for any variable. Except in this case, it's one variable associated with 4 values. It looks like this in memory:



Lists are ordered, which means we can access individual elements by their positions. In Python, as in most programming languages, the first position is 0 instead of 1.

Position	**0**	**1**	**2**	**3**
Value	90	85	92	89

Now we can refer to all 4 elements with one variable name, and square brackets to indicate position:

```
print(grades[0]) # prints 90
print(grades[1]) # prints 85
```

Common List Operations

What do we do with a list once it exists in our program? Certainly it depends on the data and your application, but here are a few common ones:

1. ***Populate a list from a file --- one value per line***
   Most of our data for variables comes from files, as we know. Most of the files we've worked with so far are in a format like this:

   ```
   93
   87
   85
   90
   100
   ```

   grades.txt — we want to make one list out of all the values

We’ll use a while loop to read from the file until we hit the end. We’ll read in each number and append it to the end of our list.

```
# make an empty list and populate with the contents of a file
grades = []
with open("grades.txt") as infile:
 while True:
   line = infile.readline()
   if line == "":
     break
   grades.append(int(line))
```

2. ***Populate a list from a file — all the values on one line***
It’s more common in data science that a bunch of related values live together, on the same line of an input file, like this:

```
93,87,85,90,100
```

grades.txt — we want to make one list out of all the values

In this case, we can read the whole line at once, and we want to use Python’s **split** function to separate each int out by itself.

```
# make an empty list and populate with the contents of a file
grades = []
with open("grades.txt") as infile:
 line = infile.readline()
 temp = line.split(",")
 for num in temp:
   grades.append(int(num))
```

3. ***Iterate over the list by value***. Once a list is populated, you might want to inspect every item in the list. In this example, we have a list full of grades, so maybe we want to average them, or just print each one out.

When we iterate **by value** we look at each value in the list without caring about the position:

```
for grade in grades:
   print(grade)
```

Note that when we iterate by value, we can’t modify the list.

4. ***Iterate over the list by position***. This is another way to look at every single element in a list. When we iterating by position, we can care about both the position of an element and its value.

When we iterate **by position** we look at each position in the list and use square brackets to pull out the value. Here, let's give everyone an extra one-point for each grade, a little bonus:

```
for i in range(len(grades)):
      grades[i] = grades[i] + 1
```

Note that when we iterate by position, we can (and often do!) modify the list.

3. ***Find out if a specific item exists in the list.*** The input came from the user in this case, and maybe we want to know if they put in a specific value. Python makes this nice and easy with the keyword *in:*

```
if 100 in grades:
      print("You got a perfect score on something!")
```

4. ***Remove an element from the list.*** Sticking with the grades example, we might want to remove the lowest value in the list (after, of course, we find the lowest value). Python allows for a few different ways to remove items from a list, but we'll stick with two of them:

```
mylist.pop(i)          # remove the item at position i

mylist.remove(80)      # remove the first item with value 80
```

More List Operations
Just as with strings, there are a ton of different things you can do with lists. In IDLE's interactive mode, type **`help(list)`** to get an idea of some of them. And/or, just try stuff out. We've got a few of the important operations below:

We use square brackets to access individual items within the list. For example, if you wanted to print the item contained at position 2, you would use this Python statement:

```
print(mylist[2])
```

Similarly, if you want to replace the value at position 2 with some other value, you would do this:

```
mylist[2] = 98
```

If you try either of these and there is no value at position 2, you'll get an *IndexError* message, because you're trying to access memory that's not part of your list.

```
len(mylist)
```

Returns the length of the list. Does not modify the list.

```
mylist.append(90)
```

Inserts the value 90 at the end of the list. The list’s length increases by one.

```
mylist.insert(2, 25)
```

Inserts the value 25 at position 2, shifting the other elements to the right by one slot to make room. The list’s length increases by one.

```
mylist.sort()
```

Sorts the elements in the list in ascending order.

```
mylist.reverse()
```

Reverses the order of the elements in the list.

```
mylist.index(90)
```

Returns the position of the first value 90. Does not modify the list.