Programming Fundamental

Lab Manual 09

Learning Outcomes:

- Students should be able to think logically and develop problem-solving skills
- Students should be able to learn how to declare variables and manipulate their values
- Students should be able to write the input and output instructions

Introduction

In this lab, you will learn to store more than one valuable in a single variable. This by itself is one of the most powerful ideas in programming, and it introduces a number of other central concepts such as loops. If this section ends up making sense to you, you will be able to start writing some interesting programs, and you can be more confident that you will be able to develop overall competence as a programmer.

Contents

Lists

Introducing Lists

Example

A list is a collection of items, that is stored in a variable. The items should be related in some way, but there are no restrictions on what can be stored in a list. Here is a simple example of a list, and how we can quickly access each item in the list.

In [2]:
```
students = ['bernice', 'aaron', 'cody']
list1= list() #this will give you an empty list.
print(students)
print(list1)
```

```
['bernice', 'aaron', 'cody']
[]
```

Naming and defining a list

Since lists are collection of objects, it is good practice to give them a plural name. If each item in your list is a car, call the list 'cars'. If each item is a dog, call your list 'dogs'. This gives you a straightforward way to refer to the entire list ('dogs'), and to a single item in the list ('dog').

In Python, square brackets [] designate a list. To define a list, you give the name of the list, the equals sign, and the values you want to include in your list within square brackets.

In [ ]:
```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
```

Accessing one item in a list

Items in a list are identified by their position in the list, starting with zero.

To access the first element in a list, you give the name of the list, followed by a zero in parentheses.

In [3]:
```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

dog = dogs[0]
print(dog)
```

```
border collie
```

The number in parentheses is called the **index** of the item. Because lists start at zero, the index of an item is always one less than its position in the list. So to get the second item in the list, we need to use an index of 1.

In [4]:
```
###highlight=[4]
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

dog = dogs[1]
print(dog)
```

```
australian cattle dog
```

Accessing the last items in a list

You can probably see that to get the last item in this list, we would use an index of 2. This works, but it would only work because our list has exactly three items. To get the last item in a list, no matter how long the list is, you can use an index of -1.

```
In [5]: ###highlight=[4]
        dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        dog = dogs[-1]
        print(dog)
```

```
labrador retriever
```

This syntax also works for the second to last item, the third to last, and so forth.

```
In [ ]: ###highlight=[4]
        dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        dog = dogs[-2]
        print(dog.title())
```

```
Australian Cattle Dog
```

You can't use a negative number larger than the length of the list, however.

```
In [ ]: ###highlight=[4]
        dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        dog = dogs[-4]
        print(dog.title())
```

```
-----------------------------------------------------------------------
----
IndexError                                Traceback (most recent call l
ast)
<ipython-input-33-32c58df001ad> in <module>()
      1 dogs = ['border collie', 'australian cattle dog', 'labrador ret
riever']
      2 
----> 3 dog = dogs[-4]
      4 print(dog.title())

IndexError: list index out of range
```

Exercises

First List

- Store the values 'python', 'c', and 'java' in a list. Print each of these values out, using their position in the list.

First Neat List

- Store the values 'python', 'c', and 'java' in a list. Print a statement about each of these values, using their position in the list.

Your First List

- Think of something you can store in a list. Make a list with three or four items, and then print a message that includes at least one item from your list.

Lists and Looping

Accessing all elements in a list

This is one of the most important concepts related to lists. You can have a list with a million items in it, and in three lines of code you can write a sentence for each of those million items. If you want to understand lists, and become a competent programmer, make sure you take the time to understand this section.

We use a loop to access all the elements in a list. A loop is a block of code that repeats itself until it runs out of items to work with, or until a certain condition is met. In this case, our loop will run once for every item in our list. With a list that is three items long, our loop will run three times.

Let's take a look at how we access all the items in a list, and then try to understand how it works.

In [ ]:
```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

for dog in dogs:
    print(dog)
```

```
border collie
australian cattle dog
labrador retriever
```

We have already seen how to create a list, so we are really just trying to understand how the last two lines work. These last two lines make up a loop, and the language here can help us see what is happening:

```
for dog in dogs:
```

- The keyword "for" tells Python to get ready to use a loop.
- The variable "dog", with no "s" on it, is a temporary placeholder variable. This is the variable that Python will place each item in the list into, one at a time.
- The first time through the loop, the value of "dog" will be 'border collie'.
- The second time through the loop, the value of "dog" will be 'australian cattle dog'.
- The third time through, "dog" will be 'labrador retriever'.
- After this, there are no more items in the list, and the loop will end.

Doing more with each item

We can do whatever we want with the value of "dog" inside the loop. In this case, we just print the name of the dog.

```
print(dog)
```

In [ ]:
```
###highlight=[5]
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

for dog in dogs:
    print('I like ' + dog + 's.')
```

```
I like border collies.
I like australian cattle dogs.
I like labrador retrievers.
```

Enumerating a list

When you are looping through a list, you may want to know the index of the current item. You could always use the *list.index(value)* syntax, but there is a simpler way. The *enumerate()* function tracks the index of each item for you, as it loops through the list:

In [ ]:
```
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

print("Results for the dog show are as follows:\n")
for index, dog in enumerate(dogs):
    place = str(index)
    print("Place: " + place + " Dog: " + dog.title())
```

```
Results for the dog show are as follows:

Place: 0 Dog: Border Collie
Place: 1 Dog: Australian Cattle Dog
Place: 2 Dog: Labrador Retriever
```

To enumerate a list, you need to add an *index* variable to hold the current index. So instead of

```
for dog in dogs:
```

You have

```
for index, dog in enumerate(dogs)
```

The value in the variable *index* is always an integer. If you want to print it in a string, you have to turn the integer into a string:

```
str(index)
```

The index always starts at 0, so in this example the value of *place* should actually be the current index, plus one:

In [ ]:
```
###highlight=[6]
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

print("Results for the dog show are as follows:\n")
for index, dog in enumerate(dogs):
    place = str(index + 1)
    print("Place: " + place + " Dog: " + dog.title())
```

```
Results for the dog show are as follows:

Place: 1 Dog: Border Collie
Place: 2 Dog: Australian Cattle Dog
Place: 3 Dog: Labrador Retriever
```

A common looping error

One common looping error occurs when instead of using the single variable *dog* inside the loop, we accidentally use the variable that holds the entire list:

In [ ]:
```
###highlight=[5]
dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

for dog in dogs:
    print(dogs)
```

```
['border collie', 'australian cattle dog', 'labrador retriever']
['border collie', 'australian cattle dog', 'labrador retriever']
['border collie', 'australian cattle dog', 'labrador retriever']
```

In this example, instead of printing each dog in the list, we print the entire list every time we go through the loop. Python puts each individual item in the list into the variable *dog*, but we never use that variable. Sometimes you will just get an error if you try to do this:

```
In [ ]: ###highlight=[5]
        dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        for dog in dogs:
            print('I like ' + dogs + 's.')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-20-8e7acc74d7a9> in <module>()
      2 
      3 for dog in dogs:
----> 4     print('I like ' + dogs + 's.')

TypeError: Can't convert 'list' object to str implicitly
```

Exercises

First List - Loop

- Repeat *First List*, but this time use a loop to print out each value in the list.

First Neat List - Loop

- Repeat *First Neat List*, but this time use a loop to print out your statements. Make sure you are writing the same sentence for all values in your list. Loops are not effective when you are trying to generate different output for each value in your list.

Your First List - Loop

- Repeat *Your First List*, but this time use a loop to print out your message for each item in your list. Again, if you came up with different messages for each value in your list, decide on one message to repeat for each value in your list.

Common List Operations

Modifying elements in a list

You can change the value of any element in a list if you know the position of that item.

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        dogs[0] = 'australian shepherd'
        print(dogs)
```

```
['australian shepherd', 'australian cattle dog', 'labrador retriever']
```

Finding an element in a list

If you want to find out the position of an element in a list, you can use the index() function.

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        print(dogs.index('australian cattle dog'))
```

```
1
```

This method returns a ValueError if the requested item is not in the list.

```
In [ ]: ###highlight=[4]
        dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        print(dogs.index('poodle'))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-a9e05e37e8df> in <module>()
      1 dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
      2 
----> 3 print(dogs.index('poodle'))

ValueError: 'poodle' is not in list
```

Testing whether an item is in a list

You can test whether an item is in a list using the "in" keyword. This will become more useful after learning how to use if-else statements.

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']

        print('australian cattle dog' in dogs)
        print('poodle' in dogs)
```

```
True
False
```

Adding items to a list

Appending items to the end of a list

We can add an item to a list using the append() method. This method adds the new item to the end of the list.

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
        dogs.append('poodle')

        for dog in dogs:
            print(dog.title() + "s are cool.")
```

```
Border Collies are cool.
Australian Cattle Dogs are cool.
Labrador Retrievers are cool.
Poodles are cool.
```

Inserting items into a list

We can also insert items anywhere we want in a list, using the **insert()** function. We specify the position we want the item to have, and everything from that point on is shifted one position to the right. In other words, the index of every item after the new item is increased by one.

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
        dogs.insert(1, 'poodle')

        print(dogs)
```

```
['border collie', 'poodle', 'australian cattle dog', 'labrador retriever']
```

Note that you have to give the position of the new item first, and then the value of the new item. If you do it in the reverse order, you will get an error.

Creating an empty list

Now that we know how to add items to a list after it is created, we can use lists more dynamically. We are no longer stuck defining our entire list at once.

A common approach with lists is to define an empty list, and then let your program add items to the list as necessary. This approach works, for example, when starting to build an interactive web site. Your list of users might start out empty, and then as people register for the site it will grow. This is a simplified approach to how web sites actually work, but the idea is realistic.

Here is a brief example of how to start with an empty list, start to fill it up, and work with the items in the list. The only new thing here is the way we define an empty list, which is just an empty set of square brackets.

In [ ]:
```python
# Create an empty list to hold our users.
usernames = []

# Add some users.
usernames.append('bernice')
usernames.append('cody')
usernames.append('aaron')

# Greet all of our users.
for username in usernames:
    print("Welcome, " + username.title() + '!')
```

```
Welcome, Bernice!
Welcome, Cody!
Welcome, Aaron!
```

If we don't change the order in our list, we can use the list to figure out who our oldest and newest users are.

In [7]:
```python
###highlight=[10,11,12]
# Create an empty list to hold our users.
usernames = []

# Add some users.
usernames.append('bernice')
usernames.append('cody')
usernames.append('aaron')

# Greet all of our users.
for username in usernames:
    print("Welcome, " + username + '!')

# Recognize our first user, and welcome our newest user.
print("\nThank you for being our very first user, " + usernames[0] + '!')
print("And a warm welcome to our newest user, " + usernames[-1] + '!')
```

```
Welcome, bernice!
Welcome, cody!
Welcome, aaron!

Thank you for being our very first user, bernice!
And a warm welcome to our newest user, aaron!
```

Note that the code welcoming our newest user will always work, because we have used the index -1. If we had used the index 2 we would always get the third user, even as our list of users grows and grows.

Sorting a List

We can sort a list alphabetically, in either order.

In [8]:
```
students = ['bernice', 'aaron', 'cody']

# Put students in alphabetical order.
students.sort()

# Display the list in its current order.
print("Our students are currently in alphabetical order.")
for student in students:
    print(student)

#Put students in reverse alphabetical order.
students.sort(reverse=True)

# Display the list in its current order.
print("\nOur students are now in reverse alphabetical order.")
for student in students:
    print(student)
```

```
Our students are currently in alphabetical order.
aaron
bernice
cody

Our students are now in reverse alphabetical order.
cody
bernice
aaron
```

sorted() vs. *sort()*

Whenever you consider sorting a list, keep in mind that you can not recover the original order. If you want to display a list in sorted order, but preserve the original order, you can use the *sorted()* function. The *sorted()* function also accepts the optional *reverse=True* argument.

```
In [9]: students = ['bernice', 'aaron', 'cody']

        # Display students in alphabetical order, but keep the original order.
        print("Here is the list in alphabetical order:")
        for student in sorted(students):
            print(student)

        # Display students in reverse alphabetical order, but keep the original o
        rder.
        print("\nHere is the list in reverse alphabetical order:")
        for student in sorted(students, reverse=True):
            print(student)

        print("\nHere is the list in its original order:")
        # Show that the list is still in its original order.
        for student in students:
            print(student)
```

```
Here is the list in alphabetical order:
aaron
bernice
cody

Here is the list in reverse alphabetical order:
cody
bernice
aaron

Here is the list in its original order:
bernice
aaron
cody
```

Reversing a list

We have seen three possible orders for a list:

- The original order in which the list was created
- Alphabetical order
- Reverse alphabetical order

There is one more order we can use, and that is the reverse of the original order of the list. The *reverse()* function gives us this order.

```
In [ ]: students = ['bernice', 'aaron', 'cody']
        students.reverse()

        print(students)
```

```
['cody', 'aaron', 'bernice']
```

Note that reverse is permanent, although you could follow up with another call to *reverse()* and get back the original order of the list.

Sorting a numerical list

All of the sorting functions work for numerical lists as well.

In [ ]:
```
numbers = [1, 3, 4, 2]

# sort() puts numbers in increasing order.
numbers.sort()
print(numbers)

# sort(reverse=True) puts numbers in decreasing order.
numbers.sort(reverse=True)
print(numbers)
```

```
[1, 2, 3, 4]
[4, 3, 2, 1]
```

In [ ]:
```
numbers = [1, 3, 4, 2]

# sorted() preserves the original order of the list:
print(sorted(numbers))
print(numbers)
```

```
[1, 2, 3, 4]
[1, 3, 4, 2]
```

In [ ]:
```
numbers = [1, 3, 4, 2]

# The reverse() function also works for numerical lists.
numbers.reverse()
print(numbers)
```

```
[2, 4, 3, 1]
```

Finding the length of a list

You can find the length of a list using the *len()* function.

In [ ]:
```
usernames = ['bernice', 'cody', 'aaron']
user_count = len(usernames)

print(user_count)
```

```
3
```

There are many situations where you might want to know how many items in a list. If you have a list that stores your users, you can find the length of your list at any time, and know how many users you have.

In [ ]:
```
# Create an empty list to hold our users.
usernames = []

# Add some users, and report on how many users we have.
usernames.append('bernice')
user_count = len(usernames)

print("We have " + str(user_count) + " user!")

usernames.append('cody')
usernames.append('aaron')
user_count = len(usernames)

print("We have " + str(user_count) + " users!")
```

```
We have 1 user!
We have 3 users!
```

On a technical note, the *len()* function returns an integer, which can't be printed directly with strings. We use the *str()* function to turn the integer into a string so that it prints nicely:

```
In [ ]: usernames = ['bernice', 'cody', 'aaron']
        user_count = len(usernames)

        print("This will cause an error: " + user_count)
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-43-92e732ef190e> in <module>()
      2 user_count = len(usernames)
      3 
----> 4 print("This will cause an error: " + user_count)

TypeError: Can't convert 'int' object to str implicitly
```

```
In [ ]: ###highlight=[5]
        usernames = ['bernice', 'cody', 'aaron']
        user_count = len(usernames)

        print("This will work: " + str(user_count))
```

```
This will work: 3
```

Exercises

Working List

- Make a list that includes four careers, such as 'programmer' and 'truck driver'.
- Use the *list.index()* function to find the index of one career in your list.
- Use the *in* function to show that this career is in your list.
- Use the *append()* function to add a new career to your list.
- Use the *insert()* function to add a new career at the beginning of the list.
- Use a loop to show all the careers in your list.

Starting From Empty

- Create the list you ended up with in *Working List*, but this time start your file with an empty list and fill it up using *append()* statements.
- Print a statement that tells us what the first career you thought of was.
- Print a statement that tells us what the last career you thought of was.

Ordered Working List

- Start with the list you created in *Working List*.
- You are going to print out the list in a number of different orders.
- Each time you print the list, use a for loop rather than printing the raw list.
- Print a message each time telling us what order we should see the list in.
    - Print the list in its original order.
    - Print the list in alphabetical order.
    - Print the list in its original order.
    - Print the list in reverse alphabetical order.
    - Print the list in its original order.
    - Print the list in the reverse order from what it started.
    - Print the list in its original order
    - Permanently sort the list in alphabetical order, and then print it out.
    - Permanently sort the list in reverse alphabetical order, and then print it out.

Ordered Numbers

- Make a list of 5 numbers, in a random order.
- You are going to print out the list in a number of different orders.
- Each time you print the list, use a for loop rather than printing the raw list.
- Print a message each time telling us what order we should see the list in.
    - Print the numbers in the original order.
    - Print the numbers in increasing order.
    - Print the numbers in the original order.
    - Print the numbers in decreasing order.
    - Print the numbers in their original order.
    - Print the numbers in the reverse order from how they started.
    - Print the numbers in the original order.
    - Permanently sort the numbers in increasing order, and then print them out.
    - Permanently sort the numbers in descreasing order, and then print them out.

List Lengths

- Copy two or three of the lists you made from the previous exercises, or make up two or three new lists.
- Print out a series of statements that tell us how long each list is.

Removing Items from a List

Hopefully you can see by now that lists are a dynamic structure. We can define an empty list and then fill it up as information comes into our program. To become really dynamic, we need some ways to remove items from a list when we no longer need them. You can remove items from a list through their position, or through their value.

Removing items by position

If you know the position of an item in a list, you can remove that item using the *del* command. To use this approach, give the command *del* and the name of your list, with the index of the item you want to move in square brackets:

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
        # Remove the first dog from the list.
        del dogs[0]

        print(dogs)
```

```
['australian cattle dog', 'labrador retriever']
```

Removing items by value

You can also remove an item from a list if you know its value. To do this, we use the *remove()* function. Give the name of the list, followed by the word remove with the value of the item you want to remove in parentheses. Python looks through your list, finds the first item with this value, and removes it.

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
        # Remove australian cattle dog from the list.
        dogs.remove('australian cattle dog')

        print(dogs)
```

```
['border collie', 'labrador retriever']
```

Be careful to note, however, that *only* the first item with this value is removed. If you have multiple items with the same value, you will have some items with this value left in your list.

```
In [ ]: letters = ['a', 'b', 'c', 'a', 'b', 'c']
        # Remove the letter a from the list.
        letters.remove('a')

        print(letters)
```

```
['b', 'c', 'a', 'b', 'c']
```

Popping items from a list

There is a cool concept in programming called "popping" items from a collection. Every programming language has some sort of data structure similar to Python's lists. All of these structures can be used as queues, and there are various ways of processing the items in a queue.

One simple approach is to start with an empty list, and then add items to that list. When you want to work with the items in the list, you always take the last item from the list, do something with it, and then remove that item. The *pop()* function makes this easy. It removes the last item from the list, and gives it to us so we can work with it. This is easier to show with an example:

```
In [ ]: dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
        last_dog = dogs.pop()

        print(last_dog)
        print(dogs)
```

```
labrador retriever
['border collie', 'australian cattle dog']
```

This is an example of a first-in, last-out approach. The first item in the list would be the last item processed if you kept using this approach. We will see a full implementation of this approach later on, when we learn about *while* loops.

You can actually pop any item you want from a list, by giving the index of the item you want to pop. So we could do a first-in, first-out approach by popping the first iem in the list:

```
In [ ]: ###highlight=[3]
        dogs = ['border collie', 'australian cattle dog', 'labrador retriever']
        first_dog = dogs.pop(0)

        print(first_dog)
        print(dogs)
```

```
border collie
['australian cattle dog', 'labrador retriever']
```

Exercises

Famous People

- Make a list that includes the names of four famous people.
- Remove each person from the list, one at a time, using each of the four methods we have just seen:
  - Pop the last item from the list, and pop any item except the last item.
  - Remove one item by its position, and one item by its value.
- Print out a message that there are no famous people left in your list, and print your list to prove that it is empty.

Slicing a List

Since a list is a collection of items, we should be able to get any subset of those items. For example, if we want to get just the first three items from the list, we should be able to do so easily. The same should be true for any three items in the middle of the list, or the last three items, or any x items from anywhere in the list. These subsets of a list are called *slices*.

To get a subset of a list, we give the position of the first item we want, and the position of the first item we do *not* want to include in the subset. So the slice *list[0:3]* will return a list containing items 0, 1, and 2, but not item 3. Here is how you get a batch containing the first three items.

```
In [10]: usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

         # Grab the first three users in the list.
         first_batch = usernames[0:3]

         for user in first_batch:
             print(user)
```

```
bernice
cody
aaron
```

If you want to grab everything up to a certain position in the list, you can also leave the first index blank:

In [11]:
```
###highlight=[5]
usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

# Grab the first three users in the list.
first_batch = usernames[:3]

for user in first_batch:
    print(user)
```

```
bernice
cody
aaron
```

When we grab a slice from a list, the original list is not affected:

In [12]:
```
###highlight=[7,8,9]
usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

# Grab the first three users in the list.
first_batch = usernames[0:3]

# The original list is unaffected.
for user in usernames:
    print(user)
```

```
bernice
cody
aaron
ever
dalia
```

We can get any segment of a list we want, using the slice method:

In [13]:
```
usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

# Grab a batch from the middle of the list.
middle_batch = usernames[1:4]

for user in middle_batch:
    print(user)
```

```
cody
aaron
ever
```

To get all items from one position in the list to the end of the list, we can leave off the second index:

In [14]:
```
usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

# Grab all users from the third to the end.
end_batch = usernames[2:]

for user in end_batch:
    print(user)
```

```
aaron
ever
dalia
```

Copying a list

You can use the slice notation to make a copy of a list, by leaving out both the starting and the ending index. This causes the slice to consist of everything from the first item to the last, which is the entire list.

```
In [ ]: usernames = ['bernice', 'cody', 'aaron', 'ever', 'dalia']

        # Make a copy of the list.
        copied_usernames = usernames[:]
        print("The full copied list:\n\t", copied_usernames)

        # Remove the first two users from the copied list.
        del copied_usernames[0]
        del copied_usernames[0]
        print("\nTwo users removed from copied list:\n\t", copied_usernames)

        # The original list is unaffected.
        print("\nThe original list:\n\t", usernames)
```

```
The full copied list:
         ['bernice', 'cody', 'aaron', 'ever', 'dalia']

Two users removed from copied list:
         ['aaron', 'ever', 'dalia']

The original list:
         ['bernice', 'cody', 'aaron', 'ever', 'dalia']
```

Exercises

Alphabet Slices

- Store the first ten letters of the alphabet in a list.
- Use a slice to print out the first three letters of the alphabet.
- Use a slice to print out any three letters from the middle of your list.
- Use a slice to print out the letters from any point in the middle of your list, to the end.

Protected List

- Your goal in this exercise is to prove that copying a list protects the original list.
- Make a list with three people's names in it.
- Use a slice to make a copy of the entire list.
- Add at least two new names to the new copy of the list.
- Make a loop that prints out all of the names in the original list, along with a message that this is the original list.
- Make a loop that prints out all of the names in the copied list, along with a message that this is the copied list.

Numerical Lists

There is nothing special about lists of numbers, but there are some functions you can use to make working with numerical lists more efficient. Let's make a list of the first ten numbers, and start working with it to see how we can use numbers in a list.

In [ ]:
```
# Print out the first ten numbers.
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for number in numbers:
    print(number)
```

```
1
2
3
4
5
6
7
8
9
10
```

The *range()* function

This works, but it is not very efficient if we want to work with a large set of numbers. The *range()* function helps us generate long lists of numbers. Here are two ways to do the same thing, using the *range* function.

In [ ]:
```
# Print the first ten numbers.
for number in range(1,11):
    print(number)
```

```
1
2
3
4
5
6
7
8
9
10
```

The range function takes in a starting number, and an end number. You get all integers, up to but not including the end number. You can also add a *step* value, which tells the *range* function how big of a step to take between numbers:

In [ ]:
```
# Print the first ten odd numbers.
for number in range(1,21,2):
    print(number)
```

```
1
3
5
7
9
11
13
15
17
19
```

If we want to store these numbers in a list, we can use the *list()* function. This function takes in a range, and turns it into a list:

In [ ]:
```
# Create a list of the first ten numbers.
numbers = list(range(1,11))
print(numbers)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

This is incredibly powerful; we can now create a list of the first million numbers, just as easily as we made a list of the first ten numbers. It doesn't really make sense to print the million numbers here, but we can show that the list really does have one million items in it, and we can print the last ten items to show that the list is correct.

In [ ]:
```
# Store the first million numbers in a list.
numbers = list(range(1,1000001))

# Show the length of the list:
print("The list 'numbers' has " + str(len(numbers)) + " numbers in it.")

# Show the last ten numbers:
print("\nThe last ten numbers in the list are:")
for number in numbers[-10:]:
    print(number)
```

```
The list 'numbers' has 1000000 numbers in it.

The last ten numbers in the list are:
999991
999992
999993
999994
999995
999996
999997
999998
999999
1000000
```

There are two things here that might be a little unclear. The expression

```
str(len(numbers))
```

takes the length of the *numbers* list, and turns it into a string that can be printed.

The expression

```
numbers[-10:]
```

gives us a *slice* of the list. The index `-1` is the last item in the list, and the index `-10` is the item ten places from the end of the list. So the slice `numbers[-10:]` gives us everything from that item to the end of the list.

The *min()*, *max()*, and *sum()* functions

There are three functions you can easily use with numerical lists. As you might expect, the *min()* function returns the smallest number in the list, the *max()* function returns the largest number in the list, and the *sum()* function returns the total of all numbers in the list.

In [ ]:
```
ages = [23, 16, 14, 28, 19, 11, 38]

youngest = min(ages)
oldest = max(ages)
total_years = sum(ages)

print("Our youngest reader is " + str(youngest) + " years old.")
print("Our oldest reader is " + str(oldest) + " years old.")
print("Together, we have " + str(total_years) + " years worth of life exp
erience.")
```

```
Our youngest reader is 11 years old.
Our oldest reader is 38 years old.
Together, we have 149 years worth of life experience.
```

Exercises

First Twenty

- Use the *range()* function to store the first twenty numbers (1-20) in a list, and print them out.

Larger Sets

- Take the *first_twenty.py* program you just wrote. Change your end number to a much larger number. How long does it take your computer to print out the first million numbers? (Most people will never see a million numbers scroll before their eyes. You can now see this!)

Five Wallets

- Imagine five wallets with different amounts of cash in them. Store these five values in a list, and print out the following sentences:
  - "The fattest wallet has $ *value* in it."
  - "The skinniest wallet has $ *value* in it."
  - "All together, these wallets have $ *value* in them."

List Comprehensions

It is good to be aware of list comprehensions, because you will see them in other people's code, and they are really useful when you understand how to use them. That said, if they don't make sense to you yet, don't worry about using them right away. When you have worked with enough lists, you will want to use comprehensions. For now, it is good enough to know they exist, and to recognize them when you see them. If you like them, go ahead and start trying to use them now.

Numerical Comprehensions

Let's consider how we might make a list of the first ten square numbers. We could do it like this:

```
In [ ]: # Store the first ten square numbers in a list.
        # Make an empty list that will hold our square numbers.
        squares = []

        # Go through the first ten numbers, square them, and add them to our lis
        t.
        for number in range(1,11):
            new_square = number**2
            squares.append(new_square)

        # Show that our list is correct.
        for square in squares:
            print(square)
```

```
1
4
9
16
25
36
49
64
81
100
```

This should make sense at this point. If it doesn't, go over the code with these thoughts in mind:

- We make an empty list called *squares* that will hold the values we are interested in.
- Using the *range()* function, we start a loop that will go through the numbers 1-10.
- Each time we pass through the loop, we find the square of the current number by raising it to the second power.
- We add this new value to our list *squares*.
- We go through our newly-defined list and print out each square.

Now let's make this code more efficient. We don't really need to store the new square in its own variable *new_square*; we can just add it directly to the list of squares. The line

```
new_square = number**2
```

is taken out, and the next line takes care of the squaring:

```
In [ ]: ###highlight=[8]
        # Store the first ten square numbers in a list.
        # Make an empty list that will hold our square numbers.
        squares = []

        # Go through the first ten numbers, square them, and add them to our lis
        t.
        for number in range(1,11):
            squares.append(number**2)

        # Show that our list is correct.
        for square in squares:
            print(square)
```

```
1
4
9
16
25
36
49
64
81
100
```

List comprehensions allow us to collapse the first three lines of code into one line. Here's what it looks like:

```
In [ ]: ###highlight=[2,3]
        # Store the first ten square numbers in a list.
        squares = [number**2 for number in range(1,11)]

        # Show that our list is correct.
        for square in squares:
            print(square)
```

```
1
4
9
16
25
36
49
64
81
100
```

It should be pretty clear that this code is more efficient than our previous approach, but it may not be clear what is happening. Let's take a look at everything that is happening in that first line:

We define a list called *squares*.

Look at the second part of what's in square brackets:

```
for number in range(1,11)
```

This sets up a loop that goes through the numbers 1-10, storing each value in the variable *number*. Now we can see what happens to each *number* in the loop:

```
number**2
```

Each number is raised to the second power, and this is the value that is stored in the list we defined. We might read this line in the following way:

squares = [raise *number* to the second power, for each *number* in the range 1-10]

Another example

It is probably helpful to see a few more examples of how comprehensions can be used. Let's try to make the first ten even numbers, the longer way:

```
In [ ]: # Make an empty list that will hold the even numbers.
        evens = []

        # Loop through the numbers 1-10, double each one, and add it to our list.
        for number in range(1,11):
            evens.append(number*2)

        # Show that our list is correct:
        for even in evens:
            print(even)
```

```
2
4
6
8
10
12
14
16
18
20
```

Here's how we might think of doing the same thing, using a list comprehension:

evens = [multiply each *number* by 2, for each *number* in the range 1-10]

Here is the same line in code:

```
In [ ]: ###highlight=[2,3]
        # Make a list of the first ten even numbers.
        evens = [number*2 for number in range(1,11)]

        for even in evens:
            print(even)
```

```
2
4
6
8
10
12
14
16
18
20
```

Non-numerical comprehensions

We can use comprehensions with non-numerical lists as well. In this case, we will create an initial list, and then use a comprehension to make a second list from the first one. Here is a simple example, without using comprehensions:

```
In [ ]: # Consider some students.
        students = ['bernice', 'aaron', 'cody']

        # Let's turn them into great students.
        great_students = []
        for student in students:
            great_students.append(student.title() + " the great!")

        # Let's greet each great student.
        for great_student in great_students:
            print("Hello, " + great_student)
```

```
Hello, Bernice the great!
Hello, Aaron the great!
Hello, Cody the great!
```

To use a comprehension in this code, we want to write something like this:

great_students = [add 'the great' to each *student*, for each *student* in the list of *students*]

Here's what it looks like:

```
In [ ]: ###highlight=[5,6]
        # Consider some students.
        students = ['bernice', 'aaron', 'cody']

        # Let's turn them into great students.
        great_students = [student.title() + " the great!" for student in students
        ]

        # Let's greet each great student.
        for great_student in great_students:
            print("Hello, " + great_student)
```

```
Hello, Bernice the great!
Hello, Aaron the great!
Hello, Cody the great!
```

Exercises

If these examples are making sense, go ahead and try to do the following exercises using comprehensions. If not, try the exercises without comprehensions. You may figure out how to use comprehensions after you have solved each exercise the longer way.

Multiples of Ten

- Make a list of the first ten multiples of ten (10, 20, 30... 90, 100). There are a number of ways to do this, but try to do it using a list comprehension. Print out your list.

Cubes

- We saw how to make a list of the first ten squares. Make a list of the first ten cubes (1, 8, 27... 1000) using a list comprehension, and print them out.

Awesomeness

- Store five names in a list. Make a second list that adds the phrase "is awesome!" to each name, using a list comprehension. Print out the awesome version of the names.

Working Backwards

- Write out the following code without using a list comprehension:

  plus_thirteen = [number + 13 for number in range(1,11)]

Strings as Lists

Now that you have some familiarity with lists, we can take a second look at strings. A string is really a list of characters, so many of the concepts from working with lists behave the same with strings.

Strings as a list of characters

We can loop through a string using a *for* loop, just like we loop through a list:

In [ ]:
```python
message = "Hello!"

for letter in message:
    print(letter)
```

```
H
e
l
l
o
!
```

We can create a list from a string. The list will have one element for each character in the string:

In [ ]:
```python
message = "Hello world!"

message_list = list(message)
print(message_list)
```

```
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

Slicing strings

We can access any character in a string by its position, just as we access individual items in a list:

In [ ]:
```
message = "Hello World!"
first_char = message[0]
last_char = message[-1]

print(first_char, last_char)
```

```
('H', '!')
```

We can extend this to take slices of a string:

In [ ]:
```
message = "Hello World!"
first_three = message[:3]
last_three = message[-3:]

print(first_three, last_three)
```

```
('Hel', 'ld!')
```

Finding substrings

Now that you have seen what indexes mean for strings, we can search for *substrings*. A substring is a series of characters that appears in a string.

You can use the *in* keyword to find out whether a particular substring appears in a string:

In [ ]:
```
message = "I like cats and dogs."
dog_present = 'dog' in message
print(dog_present)
```

```
True
```

If you want to know where a substring appears in a string, you can use the *find()* method. The *find()* method tells you the index at which the substring begins.

In [ ]:
```
message = "I like cats and dogs."
dog_index = message.find('dog')
print(dog_index)
```

```
16
```

Note, however, that this function only returns the index of the first appearance of the substring you are looking for. If the substring appears more than once, you will miss the other substrings.

In [ ]:
```
###highlight=[2]
message = "I like cats and dogs, but I'd much rather own a dog."
dog_index = message.find('dog')
print(dog_index)
```

```
16
```

If you want to find the last appearance of a substring, you can use the *rfind()* function:

In [ ]:
```
###highlight=[3,4]
message = "I like cats and dogs, but I'd much rather own a dog."
last_dog_index = message.rfind('dog')
print(last_dog_index)
```

```
48
```

Replacing substrings

You can use the *replace()* function to replace any substring with another substring. To use the *replace()* function, give the substring you want to replace, and then the substring you want to replace it with. You also need to store the new string, either in the same string variable or in a new variable.

In [ ]:
```
message = "I like cats and dogs, but I'd much rather own a dog."
message = message.replace('dog', 'snake')
print(message)
```

```
I like cats and snakes, but I'd much rather own a snake.
```

Counting substrings

If you want to know how many times a substring appears within a string, you can use the *count()* method.

In [ ]:
```
message = "I like cats and dogs, but I'd much rather own a dog."
number_dogs = message.count('dog')
print(number_dogs)
```

```
2
```

Splitting strings

Strings can be split into a set of substrings when they are separated by a repeated character. If a string consists of a simple sentence, the string can be split based on spaces. The *split()* function returns a list of substrings. The *split()* function takes one argument, the character that separates the parts of the string.

In [ ]:
```
message = "I like cats and dogs, but I'd much rather own a dog."
words = message.split(' ')
print(words)
```

```
['I', 'like', 'cats', 'and', 'dogs,', 'but', "I'd", 'much', 'rather',
'own', 'a', 'dog.']
```

Notice that the punctuation is left in the substrings.

It is more common to split strings that are really lists, separated by something like a comma. The *split()* function gives you an easy way to turn comma-separated strings, which you can't do much with in Python, into lists. Once you have your data in a list, you can work with it in much more powerful ways.

In [ ]:
```
animals = "dog, cat, tiger, mouse, liger, bear"

# Rewrite the string as a list, and store it in the same variable
animals = animals.split(',')
print(animals)
```

```
['dog', ' cat', ' tiger', ' mouse', ' liger', ' bear']
```

Notice that in this case, the spaces are also ignored. It is a good idea to test the output of the *split()* function and make sure it is doing what you want with the data you are interested in.

One use of this is to work with spreadsheet data in your Python programs. Most spreadsheet applications allow you to dump your data into a comma-separated text file. You can read this file into your Python program, or even copy and paste from the text file into your program file, and then turn the data into a list. You can then process your spreadsheet data using a *for* loop.

Other string methods

There are a number of other string methods (http://docs.python.org/3.3/library/stdtypes.html#string-methods) that we won't go into right here, but you might want to take a look at them. Most of these methods should make sense to you at this point. You might not have use for any of them right now, but it is good to know what you can do with strings. This way you will have a sense of how to solve certain problems, even if it means referring back to the list of methods to remind yourself how to write the correct syntax when you need it.

Exercises

Listing a Sentence

- Store a single sentence in a variable. Use a for loop to print each character from your sentence on a separate line.

Sentence List

- Store a single sentence in a variable. Create a list from your sentence. Print your raw list (don't use a loop, just print the list).

Sentence Slices

- Store a sentence in a variable. Using slices, print out the first five characters, any five consecutive characters from the middle of the sentence, and the last five characters of the sentence.

Finding Python

- Store a sentence in a variable, making sure you use the word *Python* at least twice in the sentence.
- Use the *in* keyword to prove that the word *Python* is actually in the sentence.
- Use the *find()* function to show where the word *Python* first appears in the sentence.
- Use the *rfind()* function to show the last place *Python* appears in the sentence.
- Use the *count()* function to show how many times the word *Python* appears in your sentence.
- Use the *split()* function to break your sentence into a list of words. Print the raw list, and use a loop to print each word on its own line.
- Use the *replace()* function to change *Python* to *Ruby* in your sentence.

Challenges

Counting DNA Nucleotides

- Project Rosalind (http://rosalind.info/problems/locations/) is a problem set (http://rosalind.info/problems/list-view/) based on biotechnology concepts. It is meant to show how programming skills can help solve problems in genetics and biology.
- If you have understood this section on strings, you have enough information to solve the first problem in Project Rosalind, Counting DNA Nucleotides (http://rosalind.info/problems/dna/). Give the sample problem a try.
- If you get the sample problem correct, log in and try the full version of the problem!

Transcribing DNA into RNA

- You also have enough information to try the second problem, Transcribing DNA into RNA (http://rosalind.info/problems/rna/). Solve the sample problem.
- If you solved the sample problem, log in and try the full version!

Complementing a Strand of DNA

- You guessed it, you can now try the third problem as well: Complementing a Strand of DNA (http://rosalind.info/problems/revc/). Try the sample problem, and then try the full version if you are successful.

top

Tuples

Tuples are basically lists that can never be changed. Lists are quite dynamic; they can grow as you append and insert items, and they can shrink as you remove items. You can modify any element you want to in a list. Sometimes we like this behavior, but other times we may want to ensure that no user or no part of a program can change a list. That's what tuples are for.

Technically, lists are *mutable* objects and tuples are *immutable* objects. Mutable objects can change (think of *mutations*), and immutable objects can not change.

Defining tuples, and accessing elements

You define a tuple just like you define a list, except you use parentheses instead of square brackets. Once you have a tuple, you can access individual elements just like you can with a list, and you can loop through the tuple with a *for* loop:

In [ ]:
```
colors = ('red', 'green', 'blue')
print("The first color is: " + colors[0])

print("\nThe available colors are:")
for color in colors:
    print("- " + color)
```

```
The first color is: red

The available colors are:
- red
- green
- blue
```

If you try to add something to a tuple, you will get an error:

```
In [ ]: colors = ('red', 'green', 'blue')
        colors.append('purple')
```

```
---------------------------------------------------------------------------
----
AttributeError                            Traceback (most recent call l
ast)
<ipython-input-37-ed1dbff53ab2> in <module>()
      1 colors = ('red', 'green', 'blue')
----> 2 colors.append('purple')

AttributeError: 'tuple' object has no attribute 'append'
```

The same kind of thing happens when you try to remove something from a tuple, or modify one of its elements. Once you define a tuple, you can be confident that its values will not change.

Using tuples to make strings

We have seen that it is pretty useful to be able to mix raw English strings with values that are stored in variables, as in the following:

```
In [ ]: animal = 'dog'
        print("I have a " + animal + ".")
```

```
I have a dog.
```

This was especially useful when we had a series of similar statements to make:

```
In [ ]: animals = ['dog', 'cat', 'bear']
        for animal in animals:
            print("I have a " + animal + ".")
```

```
I have a dog.
I have a cat.
I have a bear.
```

I like this approach of using the plus sign to build strings because it is fairly intuitive. We can see that we are adding several smaller strings together to make one longer string. This is intuitive, but it is a lot of typing. There is a shorter way to do this, using *placeholders*.

Python ignores most of the characters we put inside of strings. There are a few characters that Python pays attention to, as we saw with strings such as "\t" and "\n". Python also pays attention to "%s" and "%d". These are placeholders. When Python sees the "%s" placeholder, it looks ahead and pulls in the first argument after the % sign:

```
In [ ]: animal = 'dog'
        print("I have a %s." % animal)
```

```
I have a dog.
```

This is a much cleaner way of generating strings that include values. We compose our sentence all in one string, and then tell Python what values to pull into the string, in the appropriate places.

This is called *string formatting*, and it looks the same when you use a list:

```
In [ ]: animals = ['dog', 'cat', 'bear']
        for animal in animals:
            print("I have a %s." % animal)
```

```
I have a dog.
I have a cat.
I have a bear.
```

If you have more than one value to put into the string you are composing, you have to pack the values into a tuple:

```
In [ ]: animals = ['dog', 'cat', 'bear']
        print("I have a %s, a %s, and a %s." % (animals[0], animals[1], animals[2
        ]))
```

```
I have a dog, a cat, and a bear.
```

String formatting with numbers

If you recall, printing a number with a string can cause an error:

```
In [ ]: number = 23
        print("My favorite number is " + number + ".")
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-47-1ed2c5bb2bba> in <module>()
      1 number = 23
----> 2 print("My favorite number is " + number + ".")

TypeError: cannot concatenate 'str' and 'int' objects
```

Python knows that you could be talking about the value 23, or the characters '23'. So it throws an error, forcing us to clarify that we want Python to treat the number as a string. We do this by *casting* the number into a string using the *str()* function:

```
In [ ]: ###highlight=[3]
        number = 23
        print("My favorite number is " + str(number) + ".")
```

```
My favorite number is 23.
```

The format string "%d" takes care of this for us. Watch how clean this code is:

```
In [ ]: ###highlight=[3]
        number = 23
        print("My favorite number is %d." % number)
```

```
My favorite number is 23.
```

If you want to use a series of numbers, you pack them into a tuple just like we saw with strings:

```
In [ ]: numbers = [7, 23, 42]
        print("My favorite numbers are %d, %d, and %d." % (numbers[0], numbers[1
        ], numbers[2]))
```

```
My favorite numbers are 7, 23, and 42.
```

Just for clarification, look at how much longer the code is if you use concatenation instead of string formatting:

```
In [ ]: ###highlight=[3]
        numbers = [7, 23, 42]
        print("My favorite numbers are " + str(numbers[0]) + ", " + str(numbers[1
        ]) + ", and " + str(numbers[2]) + ".")
```

```
My favorite numbers are 7, 23, and 42.
```

You can mix string and numerical placeholders in any order you want.

```
In [ ]: names = ['eric', 'ever']
        numbers = [23, 2]
        print("%s's favorite number is %d, and %s's favorite number is %d." % (na
        mes[0].title(), numbers[0], names[1].title(), numbers[1]))
```

```
Eric's favorite number is 23, and Ever's favorite number is 2.
```

There are more sophisticated ways to do string formatting in Python 3, but we will save that for later because it's a bit less intuitive than this approach. For now, you can use whichever approach consistently gets you the output that you want to see.

Exercises

Gymnast Scores

- A gymnast can earn a score between 1 and 10 from each judge; nothing lower, nothing higher. All scores are integer values; there are no decimal scores from a single judge.
- Store the possible scores a gymnast can earn from one judge in a tuple.
- Print out the sentence, "The lowest possible score is ___, and the highest possible score is ___." Use the values from your tuple.
- Print out a series of sentences, "A judge can give a gymnast ___ points."
  - Don't worry if your first sentence reads "A judge can give a gymnast 1 points."
  - However, you get 1000 bonus internet points if you can use a for loop, and have correct grammar. (#hints_gymnast_scores)

Revision with Tuples

- Choose a program you have already written that uses string concatenation.
- Save the program with the same filename, but add *_tuple.py* to the end. For example, *gymnast_scores.py* becomes *gymnast_scores_tuple.py*.
- Rewrite your string sections using *%s* and *%d* instead of concatenation.
- Repeat this with two other programs you have already written.

```
In [ ]:
```