Readings for Unit 2

Licensing Information

The readings for 6.S090 are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. You are free to make and share verbatim copies (or modified versions) under the terms of that license.

Portions of these readings were modified or copied verbatim from the very nice book *Think Python 2e* by Allen Downey.

PDF of these readings also available to download: reading2.pdf

Table of Contents

1) Introduction

In the last set of readings, we introduced several types of Python objects, as well as models of how Python evaluates expressions and how it manages storing and looking up variables. We also introduced our first means of controlling the order of the evaluation of statements in a program through conditional execution and showed how we could re-use code by defining and calling functions.

In this reading, we will introduce and explore some new types of Python objects, and we'll see how to fit these new types into

our existing framework. We'll also introduce some very powerful new control flow mechanisms.

Before you dive into this assignment's new material, you may wish to review some of the readings and exercises from last week (and, in particular, check the results of the manual grading, including my comments which you can see on the individual assignment pages, if you have not yet.) Almost everything introduced in this reading will build on ideas from the last one.

2) Strings

In the last set of readings, we saw that we could display characters to the screen verbatim by enclosing them in quotation marks in a print statement. For example, running the code below will display `hello, python!` on the screen:

```python
print("hello, python!")
```

But at the time, we didn't talk much about what this statement actually meant in terms of our mental model of Python. In this section, we'll start to clarify this a bit by introducing a new Python type into our mental model: *strings*.

A *string* is a type that represents a sequence of characters[1]. In Python, this type is given the name `str`. It turns out, also, that it is fine to use either double quotes (`"`) *or* single quotes (`'`) to enclose strings.[2] So the Python expression `"yarn"` evaluates to a string, and so does `'twine'`.

Because strings are actually a type of Python object, it turns out that we can do more than just print them! We can, for example, store a string in a variable:

```python
nice = "This is a nice string."
```

We can think of this the same way we thought of other variable assignments:

- Python will start by evaluating the value on the right side of the `=` symbol (which, in this case, results in a string).
- It will then store this string in memory, and associate the name `nice` with it.

Much like we did with `int` and `float` objects, we can denote strings in our environment diagrams by simply writing their value, though it may be a good idea also to draw a box around a string so that it's clear that it is a single object. Running the above code snippet, for example, would result in the following environment diagram:[3]
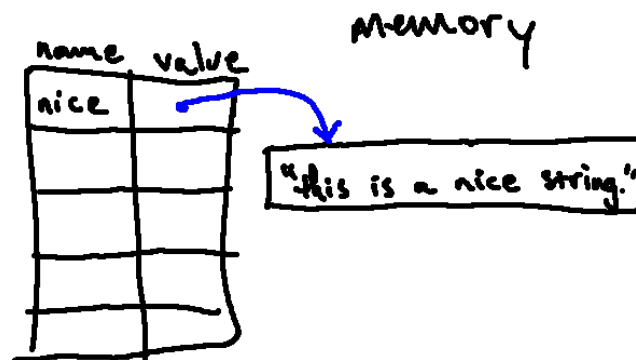
Once we have the string stored in a variable, we can include the variable in other expressions. For example, after making the definition above, we could print that string with:

```
print(nice)
```

This will look up the variable name `nice` in the global frame; doing so, it finds the string that is stored in memory, which it then displays.

Try Now:

Consider the following two small programs:

The first program reads:

```
favorite_animal = "dog"
favorite_language = "python"
print(favorite_animal)
print(favorite_language)
```

and the second reads:

```
favorite_animal = "dog"
favorite_language = "python"
print("favorite_animal")
print("favorite_language")
```

Take a close look at these programs. Syntactically, what is the difference between these two programs? How does this change affect the meaning of the program? Predict what each program will print. Then type each one into Python and run them. Do the results match your predictions?

> Show/Hide
>
> Syntactically, the only difference between the two programs is that, inside of the print statement, the second program has `"favorite_animal"` and `"favorite_language"` (with quotation marks), whereas the first program does not have them enclosed in quotation marks.
>
> Semantically, the first program will *look up the variables* called `favorite_animal` and `favorite_language`, and print the values stored in them. By contrast, the second will print the values `"favorite_animal"` and `"favorite_language"`, literally, to the screen.

2.1) Arithmetic Operations on Strings

We saw in the last set of readings that the *type* of an object is important for determining the kinds of operations we can perform on that object. For example, we could perform arithmetic with `int` and `float` objects, but not so with `NoneType` objects. Similarly, we can perform some kinds of operations on strings. Specifically, we can *concatenate* (combine) two strings together using the `+` operator and we can copy / repeat a string by using the `*` operator with an integer.

Try Now:

Try running the following in Python:

```python
print("I'm adding this string" + "to this string")
```

What value is printed? Try adding some more strings together to figure out exactly what the `+` operator does when its operands are strings.

Show/Hide

The `+` operator on strings defines *concatenation*, which is the act of joining two strings together end-to-end. The result of this operation is a *new string* which contains all of the characters in the first string, followed by all of the characters in the second.

Try Now:

Draw an environment diagram that shows the result of running the following short program, and predict what it will print:
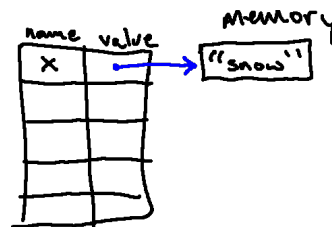
```
x = "snow"
y = "ball"

z = x + y

x = "basket" + y

print(z + " " + x)   # the middle string contains a single "space" character
```
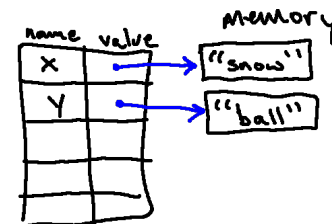
Show/Hide

After executing the first line, we have the object `"snow"` in memory, and the name `x` associated with it:

After the following line, we have a second object, `"ball"`, in memory, and the name `y` associated with it:

In the process of evaluating the third statement, Python looks up `x` (finding `"snow"`) and `y` (finding `"ball"`), and concatenates them to form a new string, `"snowball"`. This object is stored in memory, and the name `z` is associated with it:
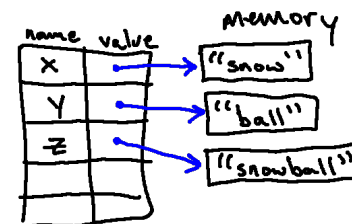
Next, we *replace* the definition of `x` with the result of evaluating `"basket" + y`. This evaluation gives us the new string `"basketball"`, which we then associate with `x`. After we do this, there are no references left to our original `"snow"` object, so it is garbage collected, giving us the following final diagram:
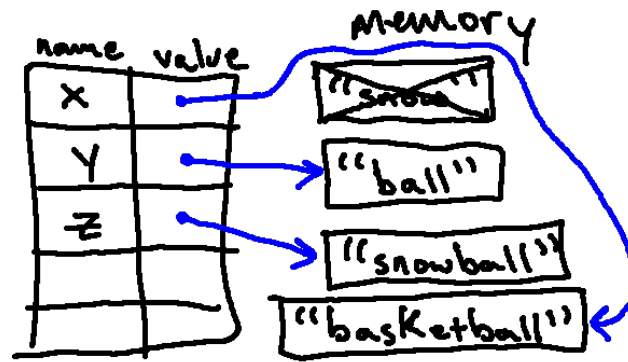
The final line contains a print statement, and we can use our substitution model to determine the value that is printed:

- z + " " + x (Loading z gives us...)
- "snowball" + " " + x (Concatenating the first two terms gives us...)
- "snowball " + x (Loading x gives us...)
- "snowball " + "basketball" (Concatenating these two strings gives us...)
- "snowball basketball"

And so the value that is printed is "snowball basketball".

Try Now:

Try to predict whether the following expressions will evaluate without errors, and, if so, try to predict the value *and type* that results from evaluating each. Then, type them into Python to check yourself. If Python generates an error message for any of these, read it carefully and try to figure out what it means and why it happened.

- `6 + 6.0`
- `"6" + "6.0"`
- `6 + "6.0"`
- `6 + "6"`

Show/Hide

- `6 + 6.0` will evaluate to a `float` with value `12.0`
- `"6" + "6.0"` will evaluate to a `str` with value `66.0` (remember that Python uses concatenation for the `+` operator applied to strings!)
- `6 + "6.0"` will result in a `TypeError`, since Python does not know how to add an `int` to a `str` (Python is not clever enough to figure out that the person writing this expression probably wanted `12.0` as a result)
- `6 + "6"` will also result in a `TypeError` for the same reason.

We got some drastically different results for the above expressions! As we saw last week, it's really important to keep track not only of the values of the objects we're working with, but of their *types* as well, since the type of the object is what defines the operations that are valid.

This also serves as a reinforcing reminder that Python is not clever about trying to figure out what we mean, and so we have to tell it things very literally and carefully.

> **Try Now:**
>
> Now let's look at the `*` operator. Try running the following code in Python:
>
> ```python
> print("cat" * 20)
> ```
>
> What is the result? What does the `*` operator do when the first argument is a string and the second is an integer?
>
> > Show/Hide
> >
> > The `*` operator concatenates *copies* of the same string (the result of the evaluating `"cat" * 20` is `"catcatcatcatcatcatcatcatcatcatcatcatcatcatcatcatcatcatcatcat"`).
>
> What happens if the second argument is not an `int`, but a `float` (i.e., `"cat" * 20.0`)? A `NoneType` (i.e., `"cat" * None`)? A `str` (i.e., `"cat" * "dog"`)? What happens if you change the order of the operands (i.e., `20 * "cat"`)?
>
> > Show/Hide
> >
> > Changing the order of the operands (i.e, `5 * "cat"`) is perfectly okay (just like with numbers, the `*` operator is commutative, so we get the same result as above).
> >
> > All of the other changes (replacing `20` with a `float`, a `NoneType`, or a `str`) give us an error message like the following:
> >
> > `TypeError: can't multiply sequence by non-int of type 'float'`
> >
> > With that message, Python is telling us that it does not know how to handle those kinds of multiplications, and rightly so! It's not clear what those expressions would mean...

2.2) String Boolean Comparisons

In last week's reading, we saw how Python can use the boolean operators we've discussed (`!=, ==, >=, <=, >, <`) to compare *numbers* with each other (and combining / comparing `int` objects with `float` objects tends to work as we would expect), but it's important to note that in Python strings can only be combined / compared with numbers under certain circumstances.

Try Now:

Try to predict whether the following expressions will evaluate without errors, and, if so, try to predict the value *and type* that results from evaluating each. Then, type them into Python to check yourself. If Python generates an error message for any of these, read it carefully and try to figure out what it means and why it happened.

- `6 == 6`
- `6 == 6.0`
- `6 > 6.0`
- `"6" == 6`
- `"6" > 6`
- `"6" != "6.0"`
- `6.0 == "6.0"`
- `"hi" == "hi"`
- `"apple" <= "banana"`
- `"A" == "a"`
- `"abcDe" < "abcda"`

Show/Hide

- `6 == 6` will evaluate to a `bool` with value `True`, since the two operands are equal.
- `6 == 6.0` will evaluate to a `bool` with value `True`, since the two operands are equal (Python knows how to compare across this particular type boundary, since `int` and `float` are so similar).
- `"6" == 6` will evaluate to a `bool` with value `False`, since one argument is a string of characters and the other is a number. Even though the string contains something that could be interpreted as a number, to Python, it is not a number (it is just a sequence of characters!).
- `"6" > 6` will result in the following error: `TypeError: '>' not supported between instances of 'str' and 'int'`. Strings and numbers can only be compared with the `==` and `!=` operators. In general, Python can check if two objects with different types are equal using `==` or check if they are not equal using `!=`, but it usually will raise an error if you try to perform the other comparisons (`<`, `>`, `>=`, `<=`).
- `"6" != "6.0"` will evaluate to a `bool` with value `True`. `!=` on strings will evaluate to `True` if and only if the two strings do not contain exactly the same characters.
- `6.0 == "6.0"` will evaluate to a `bool` with value `False`, since one argument is a string and the other is a number.
- `"hi" == "hi"` will evaluate to a `bool` with value `True`. `==` on strings will evaluate to `True` if and only if the two strings contain exactly the same characters.
- `"apple" < "banana"` will evaluate to a `bool` with value `True`. The comparison operators `<`, `>`, `>=`, `<=` can compare two strings in terms of alphabetic order (however as we will see in a moment case matters).
- `"A" == "a"` will evaluate to a `bool` with value `False`. Python considers upper / lowercase versions of the same letter as different characters. Each character in Python is represented as a number. You can check what number is associated with a character using the `ord` function. For example `ord("A")` returns 65 while `ord("a")` returns 97, meaning that to Python `"A" < "a"`. Because of this, it is important to be careful when comparing strings, especially when they include punctuation or mix upper- and lower-case letters.
- `"abcDe" < "abcda"` will evaluate to a `bool` with value `True`. While the first three characters in the string are equal, the fourth character `"D" < "d"` which makes the expression evaluate to

> True (and the fifth character is ignored).

2.3) String Methods: `upper`, `lower`, **and** `replace`

In addition to the `+` and `*` operators, python has many more useful string methods. A few functions to note are:

`.upper()`: Returns a new copy of the string with all the cased characters converted to uppercase.[4]

For example:

```
>>> 'hello123'.upper()
'HELLO123'
>>> 'this is too LOUD!'.upper()
'THIS IS TOO LOUD!'
```

`.lower()`: Returns a new copy of the string with all the cased characters converted to lowercase.

For example:

```
>>> 'NO CAPS?'.lower()
'no caps?'
>>> "WHAT??? WHY aren't HATS allowed?1?".lower()
"what??? why aren't hats allowed?1?"'
```

`.replace(old, new[, count])`: Return a new copy of the string with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

For example:

```
>>> "jar jar".replace("j", "c")
'car car'
>>> "cheese".replace("e", "o", 2)
'choose'
```

There are many more useful string methods described in the python documentation!

2.4) Converting Between Types

In the last set of readings, we saw that we could convert between `int` and `float` objects (for example, with `int(7.8)` or `float(6)`).

It is also possible to convert between strings and numeric types, provided we are dealing with strings in a particular form. For example:

- `str(6.0)` will give us the string `"6.0"`.
- `int("2")` will give us the integer `2`.

- `float("7.8")` will give us the float `7.8`.

Try Now:

What happens if you try to convert other values to integers and floats? Try, for example, the following:

- `int("tomato")`
- `int("7.8")`
- `float("6")`

Show/Hide

The first two expressions produce errors, because Python does not know how to interpret, for example, `"tomato"` as an integer or the string `"7.8"` as an integer. However, it *is* able to interpret the string `"6"` as a float: it is the float with value `"6.0"`.

---

3) Strings Are Sequences

Strings are an example of a *compound type*: they are sequences of characters. Sequences in Python have a number of interesting operations associated with them. We'll start by exploring these in the context of strings, and then generalize to other kinds of sequences.

3.1) Indexing

You can ask Python for one character from a string with the bracket operator. For example, try the following:

```
fruit = "banana"
letter = fruit[1]
print(letter)
```

The second statement selects character number 1 from `fruit`, stores it in memory, and associates the name `letter` with it. The expression inside the brackets (in this case, `1`) is called an *index*. The index, which must be an integer, indicates which character in the sequence you want.

But, running the code above, you might not get the answer you expect!

Try Now:

Run the above code in Python, note the result (which is perhaps surprising!) and continue reading.

Most people would expect character one from `"banana"` to be `"b"`. But in Python (as in many programming languages), we actually start counting at 0 rather than at 1[5].

So the indices from 0 to 5 are associated with the letters in this string as shown below:

It is perhaps also worth noting that you can also index from the *end* of a string. The index `-1` is associated with the *last* character in a string, `-2` with the next to last, and so on. So we really have two indices associated with each character:

Trying to access an index other than one of those numbers (in this case, integers between `-6` and `5`, inclusive) results in an error.

Try Now:

Try to predict whether each of the following expressions will evaluate without error, and, if so, try to predict the value of each. Once you have made your guesses, print them in Python to verify. If Python generates an error message for any of these, read it carefully and try to figure out what it means and why it happened.

- `"cat"[0]`
- `"ferret"[5]`
- `"cow"[1] == 'horse'[-4]`
- `int("60.0"[-4])`
- `'hamster'[7]`
- `"tomato"[-4]`

Show/Hide

- `"cat"[0]` will evaluate to the string `"c"`, since `"c"` is the character in position 0 in the string.
- `"ferret"[5]` will evaluate to the string `"t"`, since `"t"` is the character in position 5 in the string (`"ferret"[-1]`) would also have been `"t"`).
- `"cow"[1] == 'horse'[-4]` will evaluate to the bool `True`. `"cow[1]"` evaluates to `"o"`, and so does `'horse'[-4]`. So in the end, we compare `"o" == "o"`, which evaluates to `True`.
- `int("60.0"[-4])` will evaluate to the int `6`. `"60.0"[-4]` is the string `"6"`, which we then convert to an integer.
- `'hamster'[7]` will result in a new kind of error, an `IndexError`. The message says: `string index out of range`, which is Python's way of trying to tell us that `7` is not a valid index into the string `'hamster'`.
- `"tomato"[-4]` will evaluate to the string `'m'`, since that is the character in position `-4`.

4) Other Sequences

We will now introduce two more incredibly useful types of sequences: *tuples*[6] and *lists*.

4.1) Tuples

Tuples are sequences like strings, with the important distinction that, while strings are limited to containing only *characters*, tuples can contain *arbitrary objects*, such as integers, floats, Booleans, `None`, or even other tuples!

A tuple is specified as a comma-separated sequence of arbitrary objects, usually wrapped in parentheses. For example, the following is a tuple containing three different objects:

```
x = (7, -7.8, "blue")
```

We can perform many of the same operations on tuples that we could on strings.
For example:

- we can index into a tuple (`x[1]` gives us `-7.8`)
- we can use `+` to concatenate two tuples (`x + (1, 2, 3)` gives us `(7, -7.8, "blue", 1, 2, 3)`)

> **Try Now:**
>
> Try out some of these operations on the example tuple above, or with some tuples of your own construction.

We also need a way to represent tuples in our environment diagrams, to model how Python actually handles them in memory. We will model the above tuple `(7, -7.8, "blue")` with the following kind of drawing:
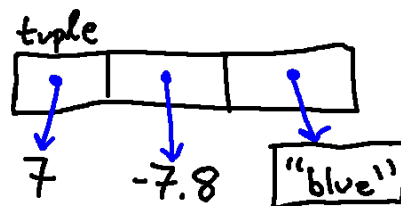
We'll draw it as a box, with the label "tuple" (so that we can keep track of types), with several references to other objects. You can think of these references as being very similar to the mappings we have already considered, from names to objects.

So after evaluating the line of code above (`x = (7, -7.8, "blue")`), we will have the following environment diagram:
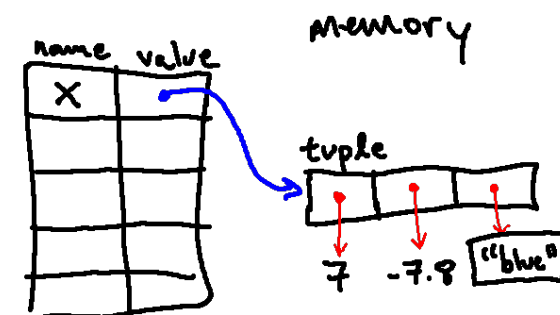
Let's examine what happens when we index into `x`. Consider, for example, running the following code:

```
    print(x[-1])
```

Python first looks up x in the global frame. Doing so, it follows the pointer from x and finds the tuple object in memory. Then, it looks up index -1 inside of x . This is the last "slot" in x , and so, following that pointer, we find the string "blue" .

Notice here that x[-1] is still a string, and so anything we can do to any other string, we can do to x[-1] . This includes *indexing into it*! So we could try the following:

```
    print(x[-1][2])
```

When evaluating x[-1][2] , Python will first look up x (finding the tuple in memory). Then it will look up index -1 inside of that tuple (finding the string "blue"). Finally, it will look up index 2 of that string (finding "u"). So this line above with print a u to the screen.

Try Now:
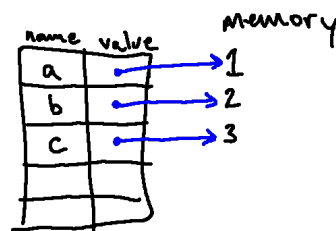
Try drawing an environment diagram for the following code:

```
a = 1
b = 2
c = 3

x = (c, b, a)
y = (3, 2, 1)
```

What is different about how the two tuples are represented in memory?
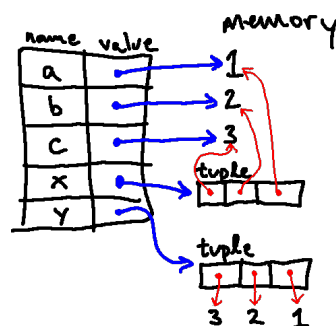
Show/Hide

After executing the first three lines, our environment diagram looks like this:

Then, when creating the first tuple, Python figures out what objects are associated with the locations in the tuple by looking up `a`, `b`, and `c`. As such, the entries in the tuple point to the *same* integer objects as `a`, `b`, and `c`:
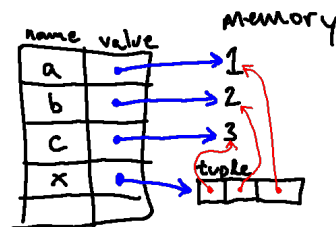
However, when creating the second tuple, Python figures out what objects are associated with the locations in the tuple by evaluating `1`, `2`, and `3`. As such, the entries in the tuple point to *different* integer objects:
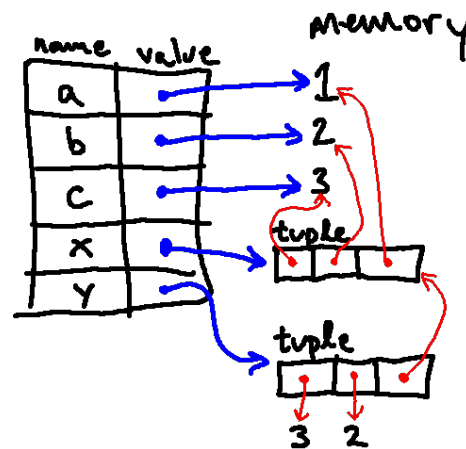
Try Now:

Note that tuples can contain *any kind of Python object*, including other tuples. So we could have had our last line instead say: `y = (3, 2, x)`. How would the final environment diagram differ if we made this change?

> Show/Hide
>
> Here is the resulting environment diagram (notice that the only change is that the last location in the `y` tuple points to the same tuple as `x`):
>
>

If we had executed this code, how would Python evaluate `y[2][0]`?

> Show/Hide
>
> Python would start by evaluating `y` and finding a tuple. It would then look up index `2` in that tuple, finding *the other tuple*, where it then looks up index `0`, finding value `3` (the same `3` that is associated with variable `c`).

4.2) Lists and Mutability

The last type of sequence we will introduce today is one of the most useful built-in types, the *list*. Lists are *almost* the same as tuples, with one exception that has big potential consequences.

Like strings or tuples, lists are sequences. Like tuples, lists can contain arbitrary Python objects. Unlike strings or tuples, however, lists are *mutable*; this means that they can be changed after they are created. In this section, we'll examine the effects of this difference.

The syntax for creating lists is similar to the syntax used for creating tuples, except that it uses square brackets instead of round brackets.

For example, we can create some lists as follows:

```
dogs = ['Lab', 'Boxer', 'Poodle']
```
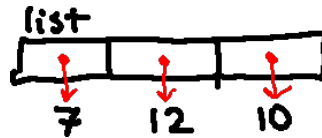
```
numbers = [42, 123]
another_list = [7, 12, 10]
empty = []   # we can also make a list that contains no elements!
```

We will represent lists in environment diagrams similarly to how we represented tuples, but we will mark them clearly as lists. For example:

With a tuple, we would get an error on the last line below:

```
my_tuple = (1,2,3)
print(my_tuple[0]) # looking up elements is fine -- no error yet
my_tuple[0] = 12
```

Specifically, we would see the error message: `TypeError: 'tuple' object does not support item assignment`.

However, if we used a list instead, we *could* modify the elements contained in the list!

Try Now:

Try running the following code:

```
dogs = ['Lab', 'Boxer', 'Poodle']
dogs[2] = 'Pincher'
print(dogs)
```

What does Python print when it executes this code?

> Show/Hide
>
> Importantly, the second line changes the value to which `dogs[2]` points (so that it now points to the string `'Pincher'` instead of to the string `'Poodle'`). So when we print `dogs`, we see:
>
> ```
> ['Lab', 'Boxer', 'Pincher']
> ```

Try Now:

Draw an environment diagram for the following code, and predict what will be displayed to the screen when the following program is run. Run your code to verify; the results may be surprising!

```
a = [7, 12, 10]
b = [4, 5, 6]

c = a

print(a)
a[0] = 8
print(a)
print(b)
b[-1] = "cow"
print(b)
print(a)
c[1] = 3.14

print(a)
```
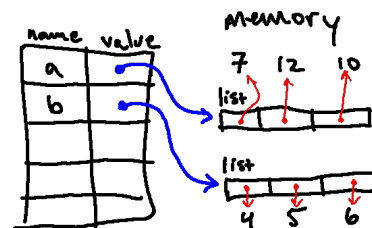
Show/Hide

After the first two lines are executed, our environment diagram should look like this:

Then, importantly, when we run the next line (`c = a`), the names `c` and `a` both refer to the *exact same list object* in memory (this does not make a copy of the list), as indicated below:
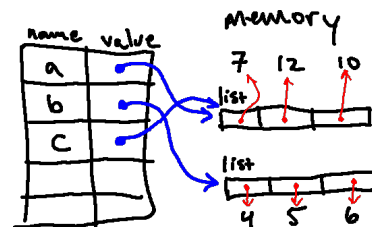
Then we print `a`, which will print the current value of `a`, which is `[7, 12, 10]`. The next line changes the value to which `a[0]` points, so we are left with:
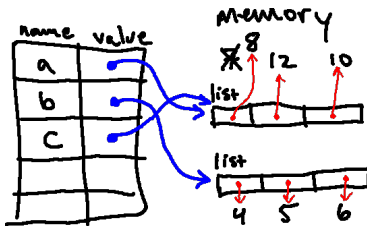
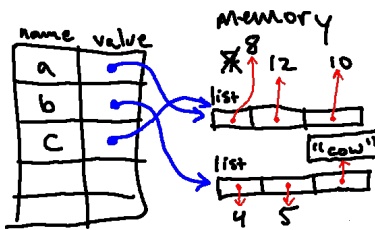Then we print `a` again, which will print the *updated* value of `a`, which is [8, 12, 10]. Then we print `b`, which will print the current value of `b`, which is [4, 5, 6]. The next line changes the value to which `b[-1]` points, so we are left with:

Then we print `b` again, which will print the *updated* value of `b`, which is [4, 5, "cow"]. Then we print `a`, which will print the current value of `a`, which is [8, 12, 10]. The next line changes the value to which `c[1]` points, so we are left with the following:
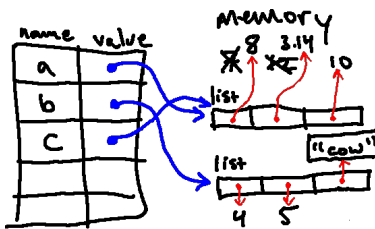
Importantly, because `a` and `c` are both associated with the *same list in memory*, looking up `a` will also see the updated value! So when we print `a`, we see [8, 3.14, 10] In the end, the whole program printed the following:

```
[7, 12, 10]
[8, 12, 10]
[4, 5, 6]
[4, 5, "cow"]
[8, 12, 10]
[8, 3.14, 10]
```

4.3) Adding Items to a List

Another common way to mutate a list is not by changing one of the elements in a list, but *adding a new element to the end of the list*. This is accomplished via `append`. For example:[7]

```
x = [5, 8, 3, 2, 1]
print(x)
```

```
    x.append(7)
    print(x)
```

This will print:

```
[5, 8, 3, 2, 1]
[5, 8, 3, 2, 1, 7]
```

Note that, concatenating two lists together like `[1, 2, 3] + [4]` creates a *new* list `[1, 2, 3, 4]` without changing the previous lists. Using the `append` method actually modifies the list in memory with which `x` is already associated. Although we sometimes have to be careful with it (because of the kinds of issues we saw above), modifying an existing list in memory is almost always substantially faster than making a new list via concatenation.

Because lists can contain arbitrary Python objects, we could use `append` to add any object to a list.

```
    x.append("a string!")
    x.append((7, 8, 9)) # a tuple
```

Try Now:

What will be printed after the following piece of code is executed?

```python
a = [1]
b = a
a.append(6)
a.append(2.0)
a.append("cat")
a[1] = "wolf"
a.append([2])
a = [4]
print(a)
print(b)
```

Show/Hide

The end result is that the following two values are printed:

```
[4]
[1, 'wolf', 2.0, 'cat', [2]]
```

In order to see why this is the case, let's simulate using an environment diagram. The first line creates a list containing a single element, a `1`, and associates the name `a` with it:
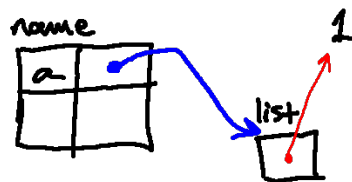
The next line associates the name `b` with *the same list in memory.*
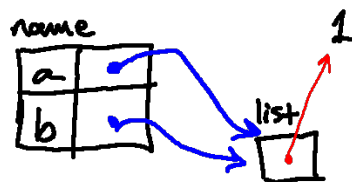
Python then looks up `a` and *modifies the list* by appending a `6` to it. Note that, because `a` and `b` are

two different names for the same object in memory, the value associated with `b` is also changing!
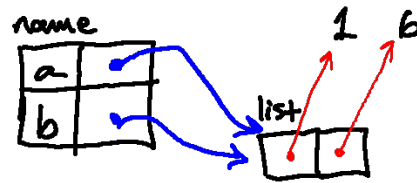
Then we append `2.0` to the same list:
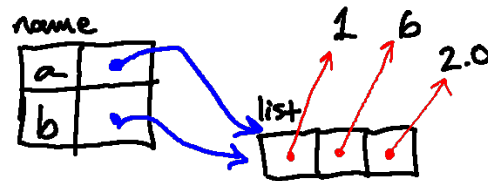
Then we append the string `"cat"`:
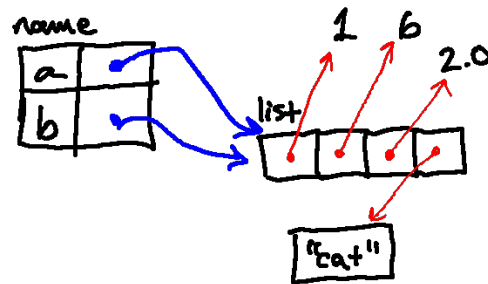
The next line then replaces the element at index `1` in the list with `"wolf"`:
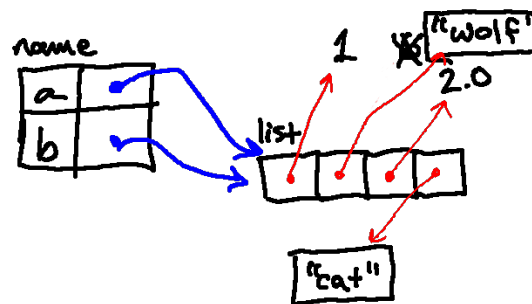
The next line then appends *a list containing the number `2`* to the list associated with the name `a`. (Notice here is an example of a list contained within another list.)
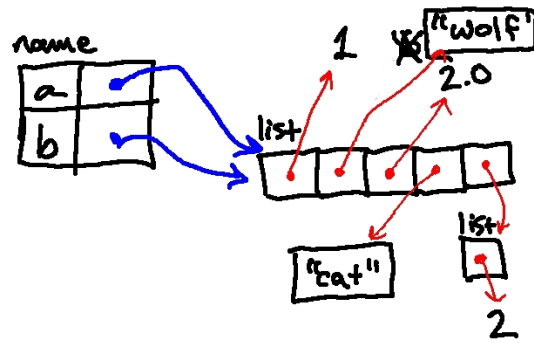
Next, we reassign `a` to be associated with a list containing a single `4`. **Note that this did not change the binding of `b`**, which is still associated with the original list.
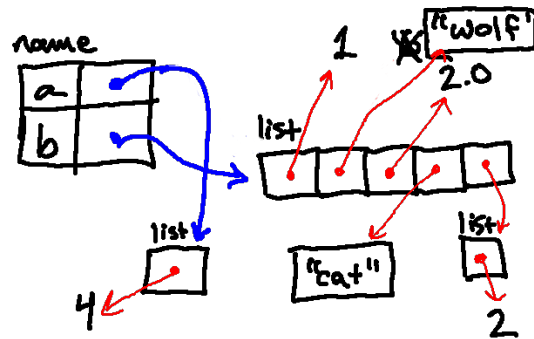
So then when we print `a`, Python follows its pointer and finds not the original list, but the new single-element list, and so it prints `[4]`. When we print `b`, Python follows its pointer and finds our long, modified list (despite the fact that we never explicitly *told* Python to do anything with `b`).

4.4) Other Common Sequence Operations

While strings, tuples, and lists have unique properties (strings only contain characters and are immutable, tuples can contain any object but are immutable, lists can contain any object and are mutable), by design they also share many similar properties (they are all ordered collections of objects), operations, and behaviors. So far, we've seen how we can:

- index into a sequence (`"hello[1]` gives us `"e"`, `(5, -7.8)[1]` gives us `-7.8`, and `[5, (4, 3), 2][1]` gives us `(4, 3)`)
- concatenate two sequences of the same type using `+` (`(('blue',) + (1, 2, 3)` gives us `("blue", 1, 2, 3))`

But we can also:

- multiply a sequence by an int using `*`. For example, `[0] * 3` will create a new list `[0, 0, 0]`.
- find the length of a sequence using `len`. For example, `len("hello")` returns

5.

- compare sequences to other objects using `==` or `!=`, which will evaluate to `True` if and only if both objects are the same type and all the corresponding elements at each index within the two objects are equal (or not equal depending on the comparison). [8]
- check if a sequence contains some value using `in` or `not in`. For example

6s.s090

https://smatz.mit.edu/6s090/week2/readings

```
>>> "h" in "horse"
True
>>> "!" not in ["a", 1, 2]
True
>>> (4, 3, 2) in (4, 3, 2)
False
```

Note that the `in` keyword is checking each individual element, not the sequence as a whole!

- cast sequences to other types. For example, `str((1,2,3))` will return the following string: `'(1, 2, 3)'`. We can also cast a string to a tuple: `tuple("abc")` creates the tuple `('a', 'b', 'c')`. `list((1, 2, 3))` creates the list: `[1, 2, 3]`. Note that unlike strings, we cannot cast a float or an int to a list or tuple. Try casting a list to a string and vice versa!
- slice into sequences to create new sequences. For example:

```
x = ["a", "b", "c", "d", "e"]
print(x[0:2])   # ["a", "b"], equivalent to x[:2]
print(x[2:5])   # ["c", "d", "e"], equivalent to x[2:] or x[-3:]
print(x[:])     # ["a", "b", "c", "d", "e"], x[0:5] creates a copy
print(x[100:])  # [] note that slicing out of bounds does not raise an error!
print(x[0:5:2]) # ["a", "c", "e"], equivalent to x[::2]
print(x[::-1])  # ["e", "d", "c", "b", "a"], reverse copy
```

Note slicing in general has a pattern of `[start_index : stop_index : step]` By default, the start_index is 0, the stop_index is the length of the sequence, and the step size is one (which is why you can omit some of the numbers and still get the same result). A step size of 2 in the second to last example means that the element at every other index between 0 <= i < 5 will be included (so 0, 2, 4). Try to experiment with slicing into strings and tuples!

There are other common sequence operations, which you can learn more about by reviewing the official Python documentation here.

5) Iteration

A lot of interesting computations on sequences involve processing them one *element* (item) at a time. Often, they start at the beginning, select each element in turn, do something to it, and continue to the end of the sequence. This pattern of processing a sequence can be referred to as *looping over* the sequence.

For example, if I wanted to display each letter in a string one at a time, I could write something like the following:

```
word = 'cat'
print(word[0])
print(word[1])
print(word[2])
print("done!")
```

which would output:

```
c
a
```

24 of 42

6/16/2023, 11:50 AM

```
t
done!
```

While this isn't too difficult for a short word, imagine trying to do this for a sentence, or a paragraph. This would likely involve copy, pasting, and modifying the same line of code over and over again, which in addition to being bug-prone is also difficult to read as well as modify.

Luckily, Python comes with some built in tools for *iteration*, which is the ability to run a block of statements repeatedly. In this section, we'll explore two such looping constructs: `while` loops and `for` loops.

5.1) While Loops

A `while` loop is a lot like a conditional, in that it consists of both a condition and a body and uses the condition to decide whether to execute the body, or to skip it. The difference is: whereas a conditional executed the body exactly once and moved on, a `while` loop will continue executing the body until the condition no longer evaluates to `True`. This pattern of flow is represented in this flow chart:
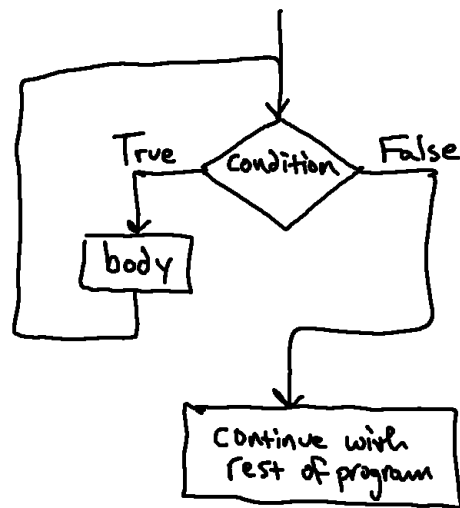
We first enter this diagram from the top. If the condition evaluates to `False`, then we skip the loop entirely and move on, but if it is `True`, we enter the body of the loop. The difference from a regular conditional is that if we do execute the body, then once we are done, we jump back and check the condition again (instead of moving on). If the condition is again `True`, we'll enter the loop again, and so on.

Consider the following example:

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

Here the condition is `n > 0` and the body of the while loop contains two lines that first display the value of n and then *decrement* it (decrease it by one). You can almost read the `while` statement as if it were English. It means "While `n` is greater than 0, display the value of `n` and then decrement `n`. When you get to 0, display the word `Blastoff!`"

Slightly more formally, here is the flow of execution for a `while` statement:

1. Determine whether the condition is true or false.
2. If false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

As written, the program above will print:

```
5
4
3
2
1
Blastoff!
```

> **Try Now:**
>
> Why was `0` not printed when the program was run? How could you modify it so that it instead printed a `0` as well, before printing blastoff?
>
> > Show/Hide
> >
> > After the last execution of the body, `1` will have just been printed to the screen and `n` will have just been decremented to `0`. Python again checks whether the condition `n>0` holds. It does not, so it moves on beyond the loop (without printing `0`).
> >
> > Changing the condition to `n>=0` would cause `0` to also be printed.

> **Try Now:**
>
> What would have been printed if we had set `n = -1` instead of `n = 5`?
>
> > Show/Hide
> >
> > If `n` had been `-1` when we first approached the loop, the condition would have evaluated to `False` that very first time. As such, we never would have entered the loop at all, and so only `Blastoff!` would have been printed.

What about our original problem of displaying every letter in a string? We could write it with a while loop as follows:

```
word = "cat"
i = 0
while i < len(word):
    print(word[i])
    i = i + 1
```

```
    print("done!")
```

While this is a bit longer than our original program, it will work for arbitrary length strings (and more general programs tend to be more useful).

5.2) Infinite Loops

It is important, when writing `while` loops, to make sure that the body of the loop changes the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise, the loop will repeat forever, which is called an *infinite loop.*[9]

In the case of the countdown program, we can prove that the loop terminates: if `n` is zero or negative, the loop never runs. Otherwise, `n` gets smaller each time through the loop, so eventually we have to get to 0.

Try Now:

What happens if you remove the line `n = n - 1` from the countdown program or change the condition to `while True:`?

For some other loops, it is not so easy to tell. For example:

```
n = 27
while n != 1:
    print(n)
    if n % 2 == 0:  # n is even
        n = n / 2
    else:  # n is odd
        n = n*3 + 1
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is `1`, which makes the condition False.

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, `n` is divided by 2. If it is odd, the value of `n` is replaced with `n*3 + 1`. For example, if `n` starts out as `3`, the resulting values of `n` are 3, 10, 5, 16, 8, 4, 2, and 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, `n` will be even every time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for *all* positive values of `n`. So far, no one has been able to prove it *or* disprove it![10]

Try Now:

What sequence of values would be printed by the above loop if we had started with `n=6`? Simulate by hand first, and then use Python to test!

Show/Hide

Technically, the values that will be printed are:

```
6
3.0
10.0
5.0
16.0
8.0
4.0
2.0
```

Why are the values after the first one floats? Because the `/` operator produces a `float`, even when its two operands are `int`s!

5.3) For loops

Oftentimes, we will want to write a traversal over a sequence. While we have seen how this can be accomplished with a `while` loop, we can do this much more concisely with a `for` loop. To get started, let's consider the following piece of code:

```python
word = "cat"
for letter in word:
    print(letter)
print('done')
```

In some ways, we can read this like English: for each letter in the string `"cat"`, print that letter (and, after that, print the word "done"). And that's exactly what Python will do. Running this code will produce the following output:

```
c
a
t
done
```

From a high-level perspective, we will repeat the code in the indented portion (the *body*) once for every element in `word`. Before each repetition, the next character in the string is assigned to the variable `letter`. Once it has gone through all characters, it will continue on with the rest of the program.

This is a much more complicated structure than some of the others we have looked at, so let's go through this code carefully and keep track of how the environment diagram changes over time.

After we execute the first line (`word = "cat"`), we have the string `"cat"` in memory, and it is associated with the name `word`:
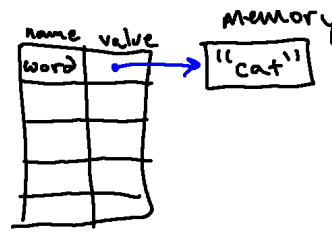
Then we enter the `for` loop. Before the first line in the body is executed, Python associates the first character from `word` with the name `letter` (just the same as if we had done `letter = word[0]`):
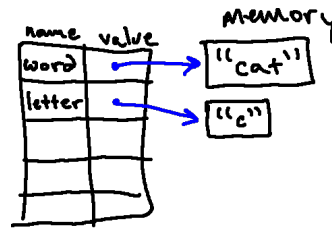
Then we run the loop body for the first time. In doing so, we execute `print(letter)`. Just like it normally would, this involves looking up the value of `letter` (which, in this case, is `"c"`), and displaying that value to the screen. So when this line executes, we see a `c` show up on the screen.

Now, since we've not reached the end of `word`, we're going to *loop again*. Before we execute the body again, Python associates the *next* character from `word` with the name `letter` (in this case, just the same as if we had done `letter = word[1]`):

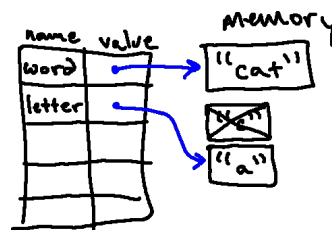Then we run the loop body a second time. In doing so, we execute `print(letter)`. Just like before, we look up the value of `letter`, but this time we get `"a"` (since `letter` was reassigned). So when this line executes this time, we see an `a` show up on the screen.

Now, since we've still not reached the end of `word`, we're going to *loop again*. Again, before we execute the body, Python associates the *next* character from `word` with the name `letter` (in this case, just the same as if we had done `letter = word[2]`):
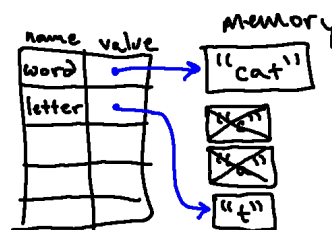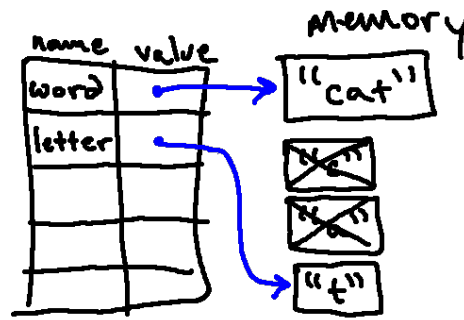
Then we run the loop body once more. In doing so, we again execute `print(letter)`. Just like before, we looking up the value of `letter`, but this time we get `"t"` (since `letter` was once again reassigned). So when this line executes this time, we see a `t` show up on the screen.

When we are finished looping, our environment diagram still looks like this:

and we return to running the rest of the program. In this case, that just means printing `'done'`. But if we had more code, we could reference `letter` (which is still bound to `"t"` after exiting the loop).

Note that, as with `if` statements, the body of a `for` loop can contain arbitrarily many expressions (and all the indented code will be executed each time through the loop).

Also note that the name `letter` was an arbitrary choice; we could use any valid variable name there. So we could have made a loop that behaves exactly the same as the one above as follows:

```python
word = "cat"
for arbitrary_variable_name in word:
    print(arbitrary_variable_name)
print('done')
```

5.4) Looping Over Integers Using `range`

In the sections above, we introduced the notion of *looping over* a sequence using the `for` keyword. In a `for` loop, we performed a computation once for every element in a sequence, setting a particular variable to point to each value in turn.

Sometimes you want to loop over particular values for which constructing a sequence would be a pain. For example, consider printing the squares of all integers from 0 to 24. One way to do this would be to write the following:

```python
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]:
    print(i**2)
```

But writing out the list of values we're looping over in this case is a real pain! Conveniently, Python offers us an easier way to build structures of this form, which we can loop over, via `range`. `range` can be used to produce a special kind of *range* object, which, while not exactly the same as a list or tuple, can serve the same purpose when used as part of a `for` loop.

To do the above more compactly, we could have written:

```python
for i in range(25):
    print(i**2)
```

If you invoke `range` with `range(4)`, then it will give you 4 elements, starting from 0 and counting upward. If you invoke it with `range(7)`, it will instead give you seven elements. When used in this way, `range` will serve the same purpose as a list containing `[0, 1, 2, 3]` or `[0, 1, 2, 3, 4, 5, 6]`.

Looping over sequence of integers is a fairly common occurrence, and so having `range` available can really save us a lot of typing (particularly if the sequence of integers you want to loop over is long!).

You can also use the `range` method to loop through all the indices in a list:

```python
my_list = [5, 3, 7]
mul_total = 0
for i in range(len(my_list)):
    mul_total += i * my_list[i]
print(mul_total) # this prints 17 because (0 * 5) + (1 * 3) + (2 * 7)
```

Try Now:

Note that like slicing, range also has a pattern of `range(start_index, stop_index, step)`, where `start_index` is by default 0, and `step` is by default 1. Try running each of the examples below to see what they will output!

```
for i in range(1, 4):
    print(i)
```

> Show/Hide
>
> This program outputs:
>
> ```
> 1
> 2
> 3
> ```
>
> Can you modify the inputs to range so that
>
> ```
> 95
> 96
> 97
> 98
> 99
> 100
> ```
>
> will print instaed?

```
for i in range(5, 0, -1):
    print(i)
```

> Show/Hide
>
> This program outputs:
>
> ```
> 5
> 4
> 3
> 2
> 1
> ```
>
> Can you modify the inputs to range so that 0 is also printed?

```
for i in range(1, 10, 2):
    print(i)
```

Show/Hide

This program outputs:

```
1
3
5
7
9
```

Can you modify the inputs to range so that

```
2
6
10
```

is printed instead?

Try Now:

Try to predict what the following program will print to the screen:

```python
count = 0
word = "cow"
print('I am thinking of a nice word.')
print('How many letters does it have?')
print(len(word))
print('I will now spell the word for you.')
for i in word:
    print(count)
    print(i)
    count = count + 1
print('The word was:')
print(word)
print('What a nice word.')
print(count)
print(i)
```

Once you have a prediction, type this code into Python and run it.

Show/Hide

This code will print:

```
I am thinking of a nice word.
How many letters does it have?
3
I will now spell the word for you.
0
c
1
o
2
w
The word was:
cow
What a nice word.
3
w
```

Now try to write this program in fewer lines of code using `range`.

Show/Hide | One way to do this would be:

```python
word = "cow"
```

```
        print('I am thinking of a nice word.')
        print('How many letters does it have?')
        print(len(word))
        print('I will now spell the word for you.')
        for count in range(len(word)):
            print(count)
            print(word[count])
        print('The word was:')
        print(word)
        print('What a nice word.')
        print(count)
        print(word[count])
```

A key insight here is that sometimes we care about both the index and the element at that index. One way to do this with a single variable is to loop over the range of the length of the sequence, which loops over the index values. Then you can use the index to access the element!

5.5) When to use `for` vs `while`?

Because anything that can be written with a for loop can be written with a while loop, it can be hard to know when to use which kind of loop. While loops have the advantage of being explicit, but for loops have the advantage of being concise, which make them easier to read. Because for loops handle the condition and incrementing the loop automatically behind the scenes, it is easier to avoid bugs that can result from forgetting to increment a variable or write the condition correctly. Additionally, it is a strong *convention* (widely used practice) among Python programmers to use for loops wherever possible. For these reasons, most of the time you should use a for loop, especially when iterating over a clearly defined sequence such as a range of numbers or elements in a list.

While loops are mostly used when we do not know what particular sequence of elements we want to iterate over, or how many times we would like to run through a loop. Sometimes, we want to repeat a sequence of statements *until a particular condition is satisfied*. An example of this will be shown in the next section where we use a while loop to approximate square roots.

5.6) While Example: Approximating Square Roots

While loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2}$$

For example, if a is 4 and x is 3:

```
a = 4
x = 3
y = (x + a/x) / 2
print(y)  # prints 2.16666666667
```

The result is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

```
x = y
y = (x + a/x) / 2
print(y)  # prints 2.00641025641
```

After a few more updates, the estimate is almost exact:

```
x = y
y = (x + a/x) / 2
print(y)  # prints 2.00001024003
```

```
x = y
y = (x + a/x) / 2
print(y)  # prints 2.00000000003
```

In general, we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
x = y
y = (x + a/x) / 2
print(y)  # prints 2.0
x = y
y = (x + a/x) / 2
print(y)  # prints 2.0
```

When `y == x`, we can stop. Here is a loop that starts with an initial estimate, `x`, and improves it until it stops changing:

```
a = 4
x = None
y = 2.5
while x != y:
    x = y
    print(x)
    y = (x + a/x) / 2
```

Note: For most values of `a` this works fine, but in general it is dangerous to test `float` equality (for some of the reasons we talked about in the last section, specifically that floats can't accurately represent all numbers!). Rather than checking whether `x` and `y` are exactly equal as above, it would be safer to loop until the difference between them of the difference between them becomes small enough (by comparing against some small error margin, for example: `while abs(x-y) > .000001:`).

5.7) For Example: Creating a list of squares

A common pattern uses `append` to build up a list of values based on some other list. For example, imagine that we had a list of integers, and we wanted to create a list of the squares of the even numbers in the original list. We could do this with, for example, the following code:

```python
original_list = [7, 4, 8, 2, 9]
new_list = []  # first make an empty list to hold the results
for element in original_list:  # for each number in the original list, do the following:
    if element % 2 == 0:  # if the number is even...
        new_list.append(element ** 2)  # add its square to the new list
print(new_list)
```

This code will proceed as follows:

- After setting `original_list` and `new_list`, Python reaches the `for` loop.
- The first time through the loop, Python sets `element` to `7` (the first element in `original_list`) and runs the loop body.
  - Because `7 % 2` is not equal to `0`, Python *does not* enter the body of the conditional; rather, it moves on.
- Now, Python reassigns `element` to the next element in the list (`4`) and enters the loop body.
  - `4 % 2 == 0` evaluates to `True`, so we enter the body of the conditional, where we add `element ** 2` (`16`) to the end of `new_list`. If we were to print `new_list` now, we would see `[16]`.
- Python continues in the same way. It reassigns `element` to the next element in the list (`8`) and enters the loop body.
  - `8 % 2 == 0` also evaluates to `True`, so we enter the body of the conditional again, where we add `64` to the end of `new_list`. If we were to print `new_list` now, we would see `[16, 64]`.
- Next, Python reassigns `element` to `2` (next in the list) and enters the loop body.
  - `2 % 2 == 0` also evaluates to `True`, so we enter the body of the conditional again, where we add `4` to the end of `new_list`. If we were to print `new_list` now, we would see `[16, 64, 4]`.
- Next, Python reassigns `element` to `9` (next in the list) and enters the loop body again.
  - Because `9 % 2` is not equal to `0`, Python *does not* enter the body of the conditional; rather, it moves on.
- Now that Python has "looped over" every element in `original_list`, it continues on past the loop to the statement: `print(new_list)`. Printing `new_list` displays the following to the screen:

```
[16, 64, 4]
```

Try Now:

Note that in general, it is a good idea to avoid adding or removing elements to a list as you are iterating over it, as this can lead to subtle bugs or unexpected results. Consider the program below. What will happen when you run it?

```python
letters = ["a", "b", "c"]
for letter in letters:
    print(letter)
    letters.append(letter)
```

Show/Hide

We get an infinite loop!

```
a
b
c
a
b
c
# continue printing same pattern forever
```

Because we keep on appending to the end of the list with each iteration of the loop, we will never reach the end of the list (eventually, the program will crash when it runs out of available memory to store this giant list!) Try printing out the length of the list in the loop to see how quickly the list can grow!

5.8) Nested Loops

Just like you can have conditional statements within conditional statements, you can have loops inside of loops!

Try Now:

What does the following program print?

```
total = 0
count = 0
nums = [[-3, 7.8, 9], [-500,100], [32, 1]]
for item in nums:
    for sub in item:
        total += sub
        count += 1

print(total / count)
```

Show/Hide

Output:

```
-50.457142857142856
```

This program iterates through each item (each list in nums) and then through each sub-item (each number in item) and finds the total and count of all the numbers to calculate the average of all the numbers in the lists.

Is there a simpler way to write this program?

Show/Hide

Yes! Using python's built-in methods len and sum (which calculates the total sum of a list or tuple of numbers) can simplify the program to a single loop. Other useful sequence methods will be in next week's reading.

```
total = 0
count = 0
nums = [[-3, 7.8, 9], [-500,100], [32, 1]]
for item in nums:
    total += sum(item)
    count += len(item)

print(total / count)
```

Try Now:

What does the following program output?

```python
my_list = [(1, 2,), True, [['inner']], None, [0], "hi", ["more", "examples"], 5, 7.5]
new_list = []
for item in my_list:
    if type(item) == list or type(item) == tuple or type(item) == str:
        for sub in item:
            new_list.append(sub)
    else:
        new_list.append(item)
print(new_list)
```

Show/Hide

Output:

`[1, 2, True, ['inner'], None, 0, 'h', 'i', 'more', 'examples', 5, 7.5]`

This is a simple example of list 'flattening.' The if statement inside the first loop checks if the item is iterable by checking the type before looping over the item and adding each sub-item to the new_list. We cannot loop over primitive types like ints, bools, floats, and None, so we append those items to the end of the new_list.

6) Debugging

In this reading, we have introduced some new structures, and started moving toward more complicated programs, which can be more difficult to think about.
In general, we can attempt to manage this complexity by trying first to break our programs down into small pieces, which can be written and tested independently of the others (this is referred to as *modular design* because we are thinking of splitting the program into separable *modules*). It is generally much easier to plan, test, and implement individual pieces as you go, rather than to spend hours writing a big program, and then find it does not work, and have to sift through all your code, trying to find the bugs.

However, even with all the clever design in the world, you will still occasionally find yourself in the (inevitable) position of having a big program with a bug in it; in that case, do not despair! Debugging a program does not require brilliance or creativity or much in the way of insight. What it requires is *persistence* and a *systematic approach*, because it requires reasoning not only about what we want, but about how Python will behave in response to our programs (this is why it's so important to have a strong mental model of Python!).

First of all, it is crucial to have a test case (a set of inputs to the program you are trying to debug) and *to know what the answer is supposed to be*, both for the overall program and for relevant intermediate values. To find a good test case, you might start with some special cases: what if the argument is 0 or the empty list? What if it is negative? Those cases might be easier to sort through first (and are also cases that can be easy to get wrong). Then try more general cases.

For most programs in this class, you should simulate your code *by hand using an environment diagram* before running it in

Python. We know this is tedious, but it really is important for helping you build a strong mental model of how Python behaves. With more experience, you will be able to make these predictions quickly in your head. But for now, *draw it out!*

Then the question remains: if your program gets your test case(s) wrong, what should you do? **Resist the temptation to start changing your program around, just to see if that will fix the problem.** Do not change any code until you know what is wrong with what you are doing now, and therefore believe that the change you make is going to correct the problem.

We have a few tools available to us already to this end, which can work reasonably well for small programs: the substitution model for expression evaluation, and environment diagrams. The act of simulating with these tools may help you find your error. It is important to remember that Python doesn't know what you *want* to do, only what you *tell it* to do, so you must be systematic when going through your code.

Sometimes, you may not be able to find your bug on paper. For those cases, the method we'll advocate centers around debugging systematically using `print` statements. It is worth noting that nowadays there exist tools other than `print` to help with debugging (logically called *debuggers*), but it is very rare even after years of experience programming that we find the need to use such a tool. In our minds, `print` is still the most straightforward, most powerful, and most general debugging tool in existence.

One good way to use `print` statements to help in debugging is to use them to display the results of intermediate steps along the way. Depending on the structure of your program, this might be: the values you are looping over (to make sure your bounds are correct), a complete solution to a subproblem, a partial solution to the overall problem. For your chosen location(s), you should print both *the quantity of interest* and *the value you expect that quantity to have*. If they are the same, it may be that that part of the code is working properly, and you can try `print`ing in other locations.[11]

One strategy here is to use a variation on binary search. Find a spot roughly halfway through your code at which you can predict the values of variables, or intermediate results your computation. Put a print statement there that lists expected as well as actual values of the variables. Run your test case, and check. If the predicted values match the actual ones, it is likely that the bug occurs after this point in the code; if they do not, then you have a bug prior to this point (of course, you might have a second bug after this point, but you can find that later). Now repeat the process by finding a location halfway between the beginning of the procedure and this point, placing a print statement with expected and actual values, and continuing. In this way you can narrow down the location of the bug. Study that part of the code and see if you can see what is wrong. If not, add some more print statements near the problematic part, and run it again.

The most important rule of debugging is: **Don't try to be smart; be systematic and indefatigable!** And don't despair!

7) Summary

In this reading, we introduced a few kinds of *compound objects* (strings, tuples, and lists) and began to expand upon the ideas introduced earlier, by introducing new ways of controlling the order in which Python executes statements (`for` and `while`), and using these to give more powerful ways to manipulate compound objects.

In this week's exercises, you'll get some practice with these new pieces, as well as some review on the older pieces.

In the next set of readings and exercises, we will introduce more tools and a powerful Python built-in type: the dictionary.

Footnotes

[1] It is called a *string* because, in some sense, the characters it contains are "strung together."

[2] Some people like to argue that using single quotes is better style, but we think either one is fine.

[3] Also note that, from here forward, our environment diagrams are likely to get a little bit more crowded. As such, we'll start leaving off the "blob" that represents memory, and simply let the open space represent memory.

[4] Explanations from python documentation

[5] This document provides a cogent argument for starting with 0.

[6] Pronounced: TOO-pullz

[7] This syntax might feel a little bit weird for now, but we will expand on it and learn what exactly it does in the coming weeks' materials.

[8] Note that like strings, we can compare tuples to other tuples and lists to other lists using other comparison operators `<, >, <=` and `>=` if the elements at corresponding indices within the objects are also comparable. So `(1, 2, 3) < (1, 2, 4)` is `True` because while the first two elements are equal, comparing the third elements `3 < 4` is `True`. However, an expression like `["a", "b", "c"] < [1, 2, 3]` will cause an error with a message like `TypeError: '<' not supported between instances of 'str' and 'int'`.

[9] Often, shampoo bottles come with directions that say: "Lather, rinse, and repeat." This is a source of amusement for some programmers; if we responded to these instructions the way Python does, we would never stop shampooing!

[10] See the Wikipedia page for the Collatz conjecture.

[11] In fact, this idea generalizes to other domains. For example, when debugging a circuit, one can use an *oscilloscope* to measure signals throughout the circuit, and so that device can serve the same purpose as a `print` statement.