Lecture 9

Loop Tasks 1: Accumulation; Loop Tasks 2: Search (and `break`); Repeating 50 Times With `range()`; Loop Tasks 3: Number-Based Repetitions; Loop Tasks 4: Simulations

1. Common Loop Tasks, Part 1: Accumulation

Accumulating sums and products

One basic use of loops is for *accumulation* processes, where individual bits of data are gathered into a whole. Perhaps the simplest case of this idea is taking the sum or product of a list of numbers.

The general strategy for accumulation is: start with an "empty accumulator", and then add on to it bit by bit.
In the present cases, that means starting by creating a variable called `running_sum` or `running_product`, initialized to `0` or `1` respectively. Then, you go through the list, and add/multiply on each successive term onto the sum/product. Like so:

In [ ]:
```
# EXAMPLE 1a: Accumulating sum/product

big_list_o_numbers = [15,72,1,84,52,48,83,26,94,58,73,95,51,73]

# Let's add'em up.
# Initialize sum to zero.
running_sum = 0

for num in big_list_o_numbers:
    running_sum = running_sum + num

print("Sum is:", running_sum)

# Let's multiply'em.
# Initialize prod to ONE.
running_prod = 1

for num in big_list_o_numbers:
    running_prod = running_prod * num

print("Product is:", running_prod)
```

You'll notice that "update assignments" like `sum = sum + num` and `prod = prod * num` are very common. Indeed, they are so common that they have shortcuts: `+=`, `-=`, `*=`, `/=`. These work as follows:

`<variable> += <value>` is equivalent to `<variable> = <variable> + <value>`

So, for example, `sum += 3` is the same as `sum = sum + 3` -- in other words, `sum += 3` adds 3 to `sum` **and then assigns that to be the new value of sum.** The other three operations work similarly.

In [ ]:
```
# EXAMPLE 1b: Average
# 1. Use += to find the average of the list of numbers.
# 2. Write it so that it still works if I add numbers to big_list.

big_list = [15,72,1,84,52,48,83,26,94,58,73,95,51,73]

big_sum = 0

for num in big_list:
    big_sum += num

average = big_sum/len(big_list)
print("Average is:", average)
```

Counting hits

Counting the number of elements in a list that meet a certain criteria is also a type of accumulation. You can keep a count variable, and then add one every time you encounter an item in your list that meets your given criteria. For example, how many words in the following list begin with the letter "b"?

In [ ]:
```
# EXAMPLE 1c: Counting "B" words
# How many words start with the letter "B"?

word_list = ["Apple", "Ball", "Animal", "Bell", "Band", "Carrot", "Angry", "Banana"
, "Bear", "Attic", "Candle", "Cup", "Beware"]

b_count = 0

for x in word_list:
    if x[0] == "B":
        b_count += 1


print("Number of B words =", b_count)
```

You can iterate through strings using a `for` loop, too, like you would with a list. Each pass through the loop will correspond to a single character. So, for example, if you wrote some code starting with

```
string_variable = "abcdef"
for letter in string_variable:
```

then `letter` would take on `"a"`, then `"b"`, etc.

Let's figure out how many times the letter "e" appears in this sentence that I am currently writing.

In [ ]:
```
# EXAMPLE 1d: How many "e"'s?

# Note the use of the triple quotes for a multi-line string!
my_sentence = "Let's figure out how many times the letter \"e\" appears in this sen
tence that I am currently writing."

e_counter = 0

for char in my_sentence:
    # letter will be "L", then "e", then "t", then "'", etc.
    if char == "e" or char == "E":
        e_counter += 1

print(e_counter, "e's in that paragraph.")
```

2. Common Loop Tasks, Part 2: Searching (and `break`)

Max/min

How do we find the maximum value appearing in a sorted list? (Python provides a function for this, but if you can't do this without using that function, you'll have a lot of difficulty in this class. And sometimes I require you not to use it.)

In [ ]:
```
# EXAMPLE 2a: Maximum

big_list = [15,72,1,84,52,48,83,26,94,58,73,95,51,73]


# First, we set largest to be the very first element.
current_largest = big_list[0]

# Then, we go through the list.  If a number is greater than "largest", we
# reassign "largest" to be that number.
for num in big_list:
    #
    # ????
    #

print(current_largest)
```

Think about how you do this as a human. It's a little tricky. What you do is, at least with a long list: start reading at the beginning, and always keep track of the largest you've seen so far, updating that when you see a larger one. That's what our program will do.

`current_largest` will keep track of the largest one we've seen so far. It's important to initialize this properly: a good choice is to initialize this to the very first element (that's kind of what a careful human would do, right?). Then, for each number in the list, you ask the question:

Is it bigger than `current_largest`?

- If it is: well, then `current_largest` should change -- it should be the number you're looking at right now.
- If it isn't ... well, carry on. (Don't do anything.)

At the end, `current_largest` will be the largest out of all the values. (And of course, all this works similarly for minimums too.)

In [ ]:
```
# EXAMPLE 2a': Maximum

big_list = [15,72,1,84,52,48,83,26,94,58,73,95,51,73]


# First, we set largest to be the very first element.
current_largest = big_list[0]

# Then, we go through the list.  If a number is greater than "largest", we
# reassign "largest" to be that number.
for num in big_list:
    #
    # Here is the fill-in:
    #
    if num > current_largest:
        current_largest = num

print(current_largest)
```

Searching

Suppose that we already have a list of data already stored, and we want to search through this list to see if a particular value is present. For this problem, it is important to understand that the answer we are trying to provide is "Yes" or "No". This means that we want a `bool` variable (recall that they are sometimes called ***flags***).

In [ ]:
```
# EXAMPLE 2b: Search

# The 30 most popular baby names ... of, like, 2014.  Sorry for not keeping my note
s current yall.
names = ["Jacob", "Sophia", "Mason", "Isabella", "William", "Emma", "Jayden",
         "Olivia", "Noah", "Ava", "Michael", "Emily", "Ethan", "Abigail",
         "Alexander", "Madison", "Aiden", "Mia", "Daniel", "Chloe", "Anthony",
         "Elizabeth", "Matthew", "Ella", "Elijah", "Addison", "Joshua", "Natalie",
"Liam", "Lily"]
# Notice that a list can extend across several lines.


search_entry = input("Input a name to search for: ")

# found is the flag.  We will set it to be true if and when we find the name;
# but at the beginning, we have not found the name yet.
found = False

for current_name in names:
    if current_name == search_entry:
        found = True
    else:
        found = False  # Hmmm...

if found == True:
    print(search_entry + " is in the list!")
else:
    print("Nobody likes your name. Choose a better name.")
```

As we discussed: *before* you've found the name, `found` remains `False`, without needing to be reassigned to be `False`; and *after* you've found the name, you wouldn't want `found` to ever go back to `False`. So the `else` block should be removed!

Also, remember that

`if found == True:`

is redundant: you can just write

`if found:`

instead, since `found == True` is just going to be evaluated, and produce the same value as `found` itself.

In [ ]:
```
# EXAMPLE 2b': Search Fixed

# The 30 most popular baby names ... of, like, 2014.  Sorry for not keeping my note
s current yall.
names = ["Jacob", "Sophia", "Mason", "Isabella", "William", "Emma", "Jayden",
         "Olivia", "Noah", "Ava", "Michael", "Emily", "Ethan", "Abigail",
         "Alexander", "Madison", "Aiden", "Mia", "Daniel", "Chloe", "Anthony",
         "Elizabeth", "Matthew", "Ella", "Elijah", "Addison", "Joshua", "Natalie",
"Liam", "Lily"]
# Notice that a list can extend across several lines.


search_entry = input("Input a name to search for: ")

# found is the flag.  We will set it to be true if and when we find the name;
# but at the beginning, we have not found the name yet.
found = False

for current_name in names:
    if current_name == search_entry:
        found = True

if found:
    print(search_entry + " is in the list!")
else:
    print("Nobody likes your name. Choose a better name.")
```

Searching with position

Now, you could imagine that you might want to be aware not just *whether* a name is in the list, but also *where* in the list your entry is. How can we do this?

There are many ways. One way (not the most "Pythonic" way, but one that can be understood without explaining new syntax) is to have a **parallel counter**. It starts at 1 (let's report a human numbering: that is, let's not use zero based indexing), and goes up by 1 each pass through the loop.

Let's add in one new feature! We probably want to stop the search at the moment we've found the entry. For this purpose, the command `break` is useful: if the line `break` is encountered in a loop, the loop immediately stops executing; execution resumes with the first statement after the end of the loop. (We could have used this in the last example, as well.)

In [ ]:
```
# EXAMPLE 2c: Search, with a parallel counter

names = ["Jacob", "Sophia", "Mason", "Isabella", "William", "Emma", "Jayden", "Olivia", "Noah", "Ava", "Michael", "Emily", "Ethan", "Abigail",
    "Alexander", "Madison", "Aiden", "Mia", "Daniel", "Chloe", "Anthony", "Elizabeth", "Matthew", "Ella", "Elijah", "Addison", "Joshua", "Natalie", "Liam", "Lily"]
# In the beginning, not found
found = False

search_entry = input("Input a name to search for: ")


# The counter starts at 0
pos = 0

for current_name in names:
    pos += 1
    # As you get the first name, count becomes 1
    # As you get the second name, count becomes 2
    # Etc.
    if current_name == search_entry:
        # If we find the name, set found to be True,
        # AND exit the loop.
        found = True
        break


if found:
    print("{0} is in the list, at position {1}".format(search_entry, pos))
else:
    print("Society thinks you have poor taste, and your child will suffer for your choices.")
```

3. Repeating 50 Times With `range()`

What if you want to print `Jerry! Hello!` fifty times? That's pretty repetitive, so it sounds like a job for a `for` loop. But wait -- we need a list (or something like it). What's the list?

That's what ranges are for. `range(50)` will create the "list" `[0, 1, 2, 3, 4, 5, ..., 49]`. (The quotation marks will be clarified in a moment.) `range`s are enormously useful -- even though you often don't do much with the actual numbers in the list, the list itself can be used to make sure that some process repeats a specified number of times.

In [ ]:
```
# EXAMPLE 3a: Jerry! Hello!

# This will print "0 Jerry! Hello!", then "1 Jerry! Hello!", etc., down to "49 Jerry! Hello!"
# Note that despite the funny numbering, that would be exactly 50 "Jerry! Hello!"s.
for i in range(50):
    print(i, "Jerry! Hello!")


# What if you didn't want the numbers "0", "1", ..., "49" printing out?
for i in range(50):
    print("Jerry! Hello!")

    # Moral: It's ok to not use the values in the list, or the variable i, in the body of the loop.
    # The list still provides a function: making sure the body runs 50 times.
```

There are three basic ways to create `ranges`, which are "lists" composed of arithmetic sequences:

- `range(<int>)` will create the list of integers starting at `0`, up to `<int>-1`. I've called the variable `<int>` to emphasize that it must be an integers. Another important thing to emphasize is that this list will *have length equal to* `<int>`*, even though it will not include* `<int>`.
- `range(<int>, <bigger int>)` with `<int> < <bigger int>` will create the list of integers starting with `<int>`, up to *but not including* `<bigger int>`. The length of this range will be equal to `<bigger int> - <int>`.
- `range(<int>, <bigger int>, <step>)` is like the above, but it will count `<step>` at a time (at least if `<step>` is positive).

In [ ]:
```
#EXAMPLE 3b: Ranges

print("range(10):")
for i in range(10):
    print(i, end = " ")
print("\n")

print("range(10, 20):")
for i in range(10, 20):
    print(i, end = " ")
print("\n")

print("range(10, 20, 2):")
for i in range(10, 20, 2):
    print(i, end = " ")
print("\n")

# It doesn't matter whether 20 would have been part of the sequence or not.
# The range will keep going up to, but not including, any number >= 20.
print("range(10, 20, 3):")
for i in range(10, 20, 3):
    print(i, end = " ")
```

Three notes:

1. You can use negative step sizes if you want to count backwards. In that case, `range(<x>, <y>, <negative step>)` will count starting with `<x>`, and ending when it reaches a number **less than or equal to** `<y>`. So in the presence of a negative step, `<x>` should really be the larger number.
2. A `range()` isn't reallllllly a `list` if we're being entirely truthful. Python does this mostly for the sake of efficiency. Here's an analogy which hints at the thought process of Python's designers: imagine that I asked you to count from 1 to 100 out loud. Would you write down those 100 numbers, and then when you're finished, read from the list? Probably not. Python creating an honest list out of 1 to 100 would be analogous to doing that; a Python range is analogous to what you actually do, which is to generate the next number on the fly.
3. But the last note won't really matter unless you try to use a `range` outside of a `for` loop -- and most of the time, when you think this is necessary, there's a better way to do what you're trying to do. That said, for the few times where you actually need a true `list` version of a `range()`, you can write, e.g., `list(range(10))`.

In [ ]:
```
#EXAMPLE 3c: Three notes

# NOTE 1: Here's how negative steps work.
print("range(10, 0, -1):")
for i in range(10, 0, -1):
    print(i, end = " ")
print("\n")

print("range(100, 0, -7):")
for i in range(100, 0, -7):
    print(i, end = " ")
print("\n####################")

# NOTE 2: A range() isn't really a list.
# Look what happens if you want to print range(50) -- not really what you wanted, r
ight?

print(range(50))
print("\n####################")

# NOTE 3: If you reallllly want your range to be a true list, surround it with "lis
t()"

print(  list(range(50)) )

# (but if you do this, you should probably ask yourself "is this necessary"? The an
swer is usually no.)
```

4. Common Loop Tasks, Part 3: Number-Based Repetitions

Saving n entries

What if you want to, say, accept a list of n names from the user? This is a different type of accumulation process. You'll need a loop that runs n times. In addition, you should also start with an empty list (write this *before* entering the loop); and the body of the loop should ask for a value, and then append it to the loop.

In [ ]:
```
# EXAMPLE 4a: Grab n names

num_names = int(input("How many names do you want to store? "))

# List of names, initialized to be empty.

name_list = []

for i in range(num_names):
    # We don't use the variable i, and that's ok! The range just ensures the correc
t number of iterations.
    # Each iteration: we GET a name, and ADD it to the list.
    current_name = input("Gimme a name: ")
    name_list.append(current_name)

print(name_list)
```

Sigma Notation

$\sum_{n=1}^{100} \sin(n^2) = ?$ Sums like this are made for `for`-loops.

In case you're uncomfortable with Σ notation, make sure that you can unravel this into "longhand" notation:

$$\sum_{n=1}^{100} \sin(n^2) = \sin(1^2) + \sin(2^2) + \sin(3^2) + \ldots + \sin(100^2)$$

For these problems, while it isn't required, I think it's much less confusing to use a `range` that matches your notation exactly: so, since the Σ notation goes from 1 to 100, I would use `range(1, 101)`.

In [ ]:
```
# Example 4b: Sigma notation
import math

sigma = 0
for n in range(1,101):
    sigma += math.sin(n**2)
print(sigma)

###
# This works too, since it has the right number of terms.
# The formula might be a bit puzzling --
# but if you look at the first and second and last terms, it appears to work.
sigma = 0
for n in range(100):
    sigma += math.sin( (n+1)**2 )
print(sigma)
```

How would we find the sum of the squares of the first 100 odd numbers? Let's do this two ways.

First method: let's write this in Σ notation, and iterate over the index of the Σ notation.

$$1^2 + 3^2 + 5^2 + \ldots + [100th\, odd\, number]^2 = \sum_{n=1}^{100} ??$$

What is $??$ Let's look at the terms $n = 1, 2, 3$ and see if we can sniff out a formula.

When $n = 1$: the first odd number is 1 (and we square that)
When $n = 2$: the second odd number is 3 (and we square that)
When $n = 3$: the third odd number is 5 (and we square that)

1, 3, 5: these numbers increase by twosies. A general principle: a function that increases by 2 everytime n goes up by 1 is a *linear function with slope 2*. So the formula for the nth odd number should be $2n + something$, and quick trial and error shows that the exact formula is $2n - 1$.

So, the nth term is $(2n-1)^2$. And the Σ notation is $\sum_{n=1}^{100} (2n-1)^2$.

In [ ]:
```
# EXAMPLE 4c: Sum of first 100 odd squares, method 1
# Iterating over the index of the Sigma notation

sum_odd_squares = 0
for n in range(1,101):
    sum_odd_squares += (2*n - 1) ** 2

print(sum_odd_squares)
```

Second method: let's make a range consisting of just the odd numbers, using a step, and iterate directly over that.

What are the odd numbers we are squaring? 1, 3, 5, ..., but where do we end? Once again, we use the formula we discovered while we were pursuing the other method: the 100th odd number is $2(100) - 1 = 199$. Therefore, our odd numbers form `range(1, 200, 2)`.

In [ ]:
```
# EXAMPLE 4d: Sum of first 100 odd squares, method 2
# Iterating directly over the odd numbers.

sum_odd_squares = 0
for odd in range(1, 200, 2):
    sum_odd_squares += odd ** 2

print(sum_odd_squares)
```

FizzBuzz

The famous interview problem: print out all the numbers from 1 to 100, **except**: any number that is a multiple of 3 should be replaced by `Fizz`; any number that is a multiple of 5 should be replaced by `Buzz`; any number that is a multiple of *both* 3 and 5 should be replaced by `FizzBuzz`.

In [ ]:
```
# EXAMPLE 4e: FizzBuzz

for n in range(1,101):
    if n % 3 == 0:
        print("Fizz")
    elif n % 5 == 0:
        print("Buzz")
    elif n % 3 == 0 and n % 5 == 0:
        print("FizzBuzz")
    else:
        print(n)

# Look closely for the error.  How do we fix it?
```

Watch out: the problem statement deliberately leads you astray. If n is a multiple of 3, you never check whether it also happens to be a multiple of 5, and so `Fizz` alone prints out, whether or not there should be a `Buzz` along with it. So, e.g., 15 prints out `Fizz` instead of `FizzBuzz`. Solution: check if n is a multiple of both first.

In [ ]:
```
# EXAMPLE 4e': FizzBuzz Corrected

for n in range(1,101):
    if n % 3 == 0 and n % 5 == 0:   # You could also check n % 15 == 0.
        print("FizzBuzz")
    elif n % 3 == 0:
        print("Fizz")
    elif n % 5 == 0:
        print("Buzz")
    else:
        print(n)

#
# Challenge: rewrite so that the code only contains the strings "Fizz" and "Buzz",
each appearing only once.
#
```