Lesson 4: Loops

In our normal everyday, we tend to repeat a lot of processes, sometimes without noticing. For example, if you want to cook a recipe, you might need to prepare ingredients by chopping them up. You chop and chop and chop until all the ingredients are the right size. At this point, you stop chopping. If we break down this chopping task into a series of three smaller steps, we have:

- An **initialization**: We're ready to cook and have a collection of ingredients we want to chop. We will start at the first ingredient.
- A **repetition**: We're chopping away. We are performing the action of chopping over and over on each of our ingredients, one ingredient at a time.
- An **end condition**: We see that we have run out of ingredients to chop, so we stop.

In programming, this process of using an initialization, repetitions, and an ending condition is called a **loop.** In a loop, we perform a process of **iteration** or *repeating tasks*. And just like we can automate chopping with a food processor, we can also automate other tasks via loops.

Programming languages like Python implement two types of iteration:

- **Indefinite iteration**, where the number of times the loop is executed depends on how many times a condition is met.
- **Definite iteration**, where the number of times the loop will be executed is defined in advance (usually based on the data set size).

> **List Comprehension:** Python list comprehensions provide a concise way for creating lists. It consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses: `[EXPRESSION for ITEM in LIST <if CONDITIONAL>]`.

The expressions can be anything - any kind of object can go into a list.

A list comprehension always returns a list.

There are two kinds of loops we'll be discussing during this bootcamp: `for` and `while`.

`For` Loops

To begin with, let's talk about `for` loops.

In Python, a `for` loop can be used to iterate over a list of items and perform a set of actions on each item. The syntax of a `for` loop consists of assigning a temporary value to a variable on each successive iteration. When writing a for `loop`, remember to properly indent each action, otherwise an `IndentationError` will result.

In a `for` loop, we'll know in advance how many times the loop will need to iterate because we'll be working on a collection with a predefined length. We'll be using lists as our collecetion of elements in the next few

examples. With `for` loops, on each iteration, we will be able to perform an action on each element of the collection.

The general structure of a `for` loop is as follows:

In [ ]:
```
for <temporary variable> in <collection>:
  <action>
```

1. A `for` keyword indicates the start of a `for` loop.
2. A `<temporary variable>` that is used to represent the value of the element in the collection the loop is currently on.
3. An `in` keyword separates the temporary variable from the collection used for iteration.
4. A `<collection>` to loop over. In our examples, we will be using a list.
5. An `<action>` to do anything on every iteration of the loop.

Temporary Variables

A temporary variable's name is arbitrary and does not need to be defined beforehand. Both of the following code snippets do the exact same thing, regardless of the temporary variable's name:

In [4]:
```
ingredients = ['flour', 'milk', 'eggs', 'sugar', 'salt']

for i in ingredients:
  print(i)
```

```
flour
milk
eggs
sugar
salt
```

In [5]:
```
for item in ingredients:
  print(item)
```

```
flour
milk
eggs
sugar
salt
```

Programming best practices suggest that temporary variables be as descriptive as possible. Since each iteration (*step*) of the loop is accessing an ingredient, it makes more sense to call the temporary variable `ingredient` rather than `i` or `item`. Comments are also reccomennded to ensure clarity within a script or program.

Indentation

Note that in all of these examples the `print` statement is indented. Everything at the same level of indentation after the `for` loop declaration is included in the loop body and is run on every iteration of the loop. This is the same as the case of `if-else` statements defined in a previous lesson.

If you ever forget to indent, you'll get an `IndentationError` or, perhaps worse, *unexpected behavior*. So watch your indentations carefully.

In [26]:
```
for ingredient in ingredients:
    # Note the indentation
    # Any code at this level of indentation will run on each iteration of the loop
print(ingredient)
```

```
  Cell In[26], line 4
    print(ingredient)
    ^
IndentationError: expected an indented block after 'for' statement on line 1
```

Elegance in Programming

As an aside, Python loves to help us write elegant code, so it allows us to write simple `for` loops in one-line.

> **Note:** One-line `for` loops are useful for simple programs. I don't recommended you write one-line loops for any loop that has to perform multiple complex actions on each iteration. Doing so will hurt the readability of your code and may ultimately lead to buggier code and a headache-inducing debugging session (or five).

Here is the previous example in a single line:

In [27]:
```
for ingredient in ingredients: print(ingredient)
```

```
flour
milk
eggs
sugar
salt
```

> **Note:** If you're not sure how many times you should run a `for` loop, Python has a built-in function (like `print()` is a built-in function) to compute the length of a string (or list, tuple, or any other sequence). To find out the length of a sequence, simply use it as an argument to the `len()` function.

In [6]:
```
len(ingredients)
```

Out[6]: 5

Iterators

In previous examples, we iterated over a sequence (a list of ingredients). A sequence is one of many iterable objects, called **iterables**. Under the hood, the Python interpreter actually converts an iterable to an iterator. An **iterator** is a special object that gives values in succession. A major difference between a sequence and an iterator is that you cannot index an iterator. This seems like a trivial difference, but iterators make for more efficient computing than directly using a sequence with indexing.

We can explicitly convert a sequence to an iterator using the built-in function `iter()`, but wen't bother with that here because the Python interpreter automatically does this for you when you use a sequence in a loop.

> **Note:** When the list is converted to an iterator, a copy is made of each element so the original list is unchanged.

Instead, now we'll explore how to create useful iterators using the `range()`, `enumerate()`, and `zip()` built-in functions of Python. We haven't covered functions yet (stay patient, it's coming), but the syntax isn't so complicated that you won't be able to understand what these functions are doing, just like with the `print()` and `len()` pre-built functions.

`range()` function

The `range()` function gives an iterable that enables counting.

In [7]:
```
for i in range(10):
    print(i, end='  ')
```

```
0  1  2  3  4  5  6  7  8  9
```

Note that `range(10)` gives us ten numbers, from 0 to 9. As with indexing, `range()` inclusively starts at zero by default, and the ending is exclusive.

It turns out that the arguments of the `range()` function work much like indexing. If you have a single argument, you get that many integers, starting at 0 and incrementing by one. If you give two arguments, you start inclusively at the first and increment by one ending exclusively at the second argument. Finally, you can specify a stride with the third argument.

In [10]:
```
# Print numbers 2 through 9
for i in range(2, 10):
    print(i, end='  ')
```

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

In [11]:
```
# Print even numbers, 2 through 8
for i in range(2, 10, 2):
    print(i, end='     ')
```

```
2     4     6     8
```

It's often useful to make a list or tuple that has the same entries that a corresponding range object would have. This can be done via type conversion.

In [12]:
```
list(range(10))
```

Out[12]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [13]:
```
my_integers = [1, 2, 3, 4, 5]

# Since len(my_integers) = 5, this takes i from 0 to 4, exactly the number of indices (or length)
for i in range(len(my_integers)):
    my_integers[i] *= 2

my_integers
```

Out[13]:
```
[2, 4, 6, 8, 10]
```

`enumerate()` function

Let's say we want to print the indices of all `G` bases in a DNA sequence.

In [15]:
```
n_gc = 0 # Initialize GC counter
len_seq = 0 # Initialized sequence length
seq = 'GACAGACUCCAUGCACGUGGGUAUCUGUC' # The sequence we want to analyze

# Loop through sequence and print index of G's
for base in seq:
    if base in 'Gg':
        print(len_seq, end='  ')
    len_seq += 1
```

```
0  4  12  16  18  19  20  26
```

This isn't so bad, but there's an easier way to do it. The `enumerate()` function gives an iterator that provides both the index and the item of a sequence. This means the `enumerate()` function would allow us to use an index and a base at the same time.

It's best demonstrated in practice.

In [16]:
```
# Loop through sequence and print index of G's
for i, base in enumerate(seq):
    if base in 'Gg':
        print(i, end='  ')
```

```
0  4  12  16  18  19  20  26
```

To make it clearer, we can print the index and base type for each base in the sequence.

In [17]:
```
# Print index and identity of bases
for i, base in enumerate(seq):
    print(i, base)
```

```
0 G
1 A
2 C
3 A
4 G
5 A
6 C
7 U
8 C
9 C
10 A
11 U
12 G
13 C
14 A
15 C
16 G
17 U
18 G
19 G
20 G
21 U
22 A
23 U
24 C
25 U
26 G
27 U
28 C
```

The `enumerate()` function is very useful and should be used in favor of just doing indexing. For example, many programmers, especially those first trained in lower-level languages, would write the above code similar to how we did but with the `range()` and `len()` functions, but this is not good practice in Python.

In [18]:
```
# list doubling code
my_integers = [1, 2, 3, 4, 5]

# Double each one
for i, _ in enumerate(my_integers):
    my_integers[i] *= 2

# Check out the result
my_integers
```

Out[18]:
```
[2, 4, 6, 8, 10]
```

`enumerate()` is more generic and the overhead for returning a reference to an object isn't an issue. The `range(len())` construct would break on an object without support for `len()`. In addition, you're more likely to introduce bugs by imposing indexing on objects that are iterable but not unambiguously indexable. So, it's better to use `enumerate()`.

> ** No:** Sometimesed the undersco character `,_` is used in a variable name to denote it , as a throwaway variable thaisn't actually usedse. There is no rule for this, but this is generally accepted Python syntax and helps signal thathe underscored variable isn't going to be used.e.

One last gotcha:youf we tried to do a similar technique with a striyou'd, we ge`t a TypeE`rror because a string is immutable.

In [19]:
```
# Try to convert capital G to lower g
for i, base in enumerate(seq):
    if base == 'G':
        seq[i] = 'g'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[19], line 4
      2 for i, base in enumerate(seq):
      3     if base == 'G':
----> 4         seq[i] = 'g'

TypeError: 'str' object does not support item assignment
```

`zip()` function

The `zip()` function enables us to iterate over several iterables at once. For example, let's iterate over the jersey numbers, names, and positions of the players on the US women's national soccer team who were named to the best eleven of the 2019 World Cup in France.

In [20]:
```
names = ('Dunn', 'Ertz', 'Lavelle', 'Rapinoe')
positions = ('D', 'MF', 'MF', 'F')
numbers = (19, 8, 16, 15)

for num, pos, name in zip(numbers, positions, names):
    print(num, name, pos)
```

```
19 Dunn D
8 Ertz MF
16 Lavelle MF
15 Rapinoe F
```

`reversed()` function

This function is useful for giving an iterator that goes in the reverse direction. Instead of incrementing up through a list or sequence, we'd start at the end and work our way down. This will come in handy in the next lesson.

In [21]:
```
# NASA count down script
count_up = ('ignition', 1, 2, 3, 4, 5, 6, 7, 8 ,9, 10)

for count in reversed(count_up):
    print(count)
```

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

`While` Loops

The `for` loop is very powerful and allows us to construct iterative calculations. But when we use a `for` loop, we need to set up an iterator. A `while` loop, on the other hand, allows iteration until a conditional expression evaluates `False`.

Specifically, in Python, a `while` loop will repeatedly execute a code block as long as a condition evaluates to `True`. The condition of a `while` loop is always checked first before the block of code runs. If the condition is not initially met, then the code block will never run.

> **Note:** A *codon* roughlys means the three consecutive bases that code for an amino acid. That means that for a three-base sequence to be a codon, it must be in-register with the start codon.
>
> We're being a bit looser with the definition here and taking *codon* to mean any three consecutive bases for ease of programming.

In [22]:
```
seq = 'GACAGACUCCAUGCACGUGGGUAUCUGUC' # The sequence we want to analyze
start_codon = 'AUG' # define start codon (piece of sequence we're looking for)
i = 0 # Initialize sequence index

# Scan sequence by three's until we hit the start codon
while seq[i:i+3] != start_codon:
    i += 1
```

```
# Show the result
print('The start codon starts at index', i)
```

```
The start codon starts at index 10
```

Let's walk through this `while` loop. The value of `i` (temporary variable) is changing with each iteration, incrementing by one. Each time we consider doing another iteration, the conditional is checked: do the next three bases match the start codon? We set up the conditional to evaluate to `True` when the bases are not the start codon, so the iteration continues. In other words, iteration continues in a `while` loop until the conditional returns `False`.

> **Note:** We sliced the string the same way we sliced lists and tuples. In the case of strings, a slice gives another string, i.e., a sequential collection of characters.

Let's try looking for another codon... Except let's actually not do that. If you run the code below, it'll run forever and nothing will ever print to the screen.

In [ ]:
```
# Define codon of interest
codon = 'GCC'

# Initialize sequence index
i = 0

# Scan sequence until we hit the start codon, but DON'T DO THIS!!!!!
while seq[i:i+3] != codon:
    i += 1

# Show the result
print('The codon starts at index', i)
```

The reason this runs forever is that the conditional expression in the `while` statement never returns `False`. If we slice a string beyond the length of the string the result is an empty string. Go figure.

In [23]:
```
seq[100:103]
```

Out[23]:
```
''
```

> **Infinite Loop:** An infinite loop is a loop that never terminates. Infinite loops result when the conditions of the loop prevent it from terminating. This could be due to a typo in the conditional statement within the loop or incorrect logic. To interrupt a Python program that is running forever, press the `Ctrl + C` keys together on your keyboard.

`for` vs `while`

Most anything that requires a loop can be done with either a `for` loop or a `while` loop, but there's a general rule of thumb for which type of loop to use. If you know how many times you have to do something (or if your program knows), use a `for` loop. If you don't know how many times the loop needs to run until you run it, use a `while` loop. For example, when we want to do something with each character in a string or each entry in a list, the program knows how long the sequence is and a `for` loop is more appropriate. In the previous examples in this lesson, we don't know how long it will be before we hit the start codon; it depends on the sequence you put into the program. That makes it more suited to a `while` loop.

`break` and `continue`

Iteration stops in a `for` loop when the iterator is exhausted. It stops in a `while` loop when the conditional evaluates to `False`. These is another way to stop iteration: the `break` keyword. Whenever `break` is encountered in a `for` or `while` loop, the iteration halts and execution continues outside the loop. Similarly, the `continue` keyword to stop the current iteration and proceed to the next loop iteration.

`break` and `continue` are both keywords you should keep in mind when debugging code containing loops.

> `break` **Keyword:** In a loop, the `break` keyword escapes the loop, regardless of the iteration number. Once `break` executes, the program will continue to execute after the loop.
>
> `continue` **Keyword:** In Python, the `continue` keyword is used inside a loop to skip the remaining code inside the loop code block and begin the next loop iteration.

As an example, we'll do the calculation above with a `for` loop with a `break` instead of a `while` loop.

In [24]:
```
start_codon = 'AUG'  # define start codon (piece of sequence we're looking for)

# Scan sequence until we hit the start codon
for i in range(len(seq)):
    if seq[i:i+3] == start_codon:
        print('The start codon starts at index', i)
        break
else:
    print('Codon not found in sequence.')
```

```
The start codon starts at index 10
```

Notice that we have an `else` block after the `for` loop. In Python, `for` and `while` loops can have an `else` statement after the code block to be evaluated in the loop. The contents of the `else` block are evaluated if the loop completes without encountering a `break`. This is useful for when a `while` check condition is never encountered. So instead of an infinite loop, you can have a custom error print to screen, as is the case in the above example.

> **Nested Loops:** In Python, loops can be nested inside other loops. Nested loops can be used to access items of lists which are inside other lists. The item selected from the outer loop can be used as the list for the inner loop to iterate over.

Proceed to Assignment 4.