Welcome to Tutorial 1!

In the previous tutorial, you learned all about data types: including strings and lists. In this tutorial, you will start learning about conditional statements and loops.

A quick note

You may have noticed that in some cells with Python code, there is green text that doesn't seem to do anything! Those pieces of green text are called **comments.** You can tell Python that you want to write a comment by using a pound sign (#) at the beginning of a line. Python will ignore the text that comes after the pound sign. Comments are used to tell other human readers what your code does. It's good practice to comment often, so other people trying to read and use your code are not confused!

Part 0: Conditional Statements

By now, you may have realized that writing in Python is a lot like writing in English. We've learned about variables, which are like nouns. But there's not much that we can do with just *nouns*! We need to be able to write sentences if we're going to say anything interesting.

Luckily, there are special "sentences" that we can write in Python. The first one of these is an "if" statement. For many English speakers, the "if...then" sentence is familiar to us. Here's an example:

If it is raining outside, then I will not go to the store.

So, there are two possibilities here. Either it's raining outside and I will not go to the store, or else it is sunny outside and I will (probably) go to the store.

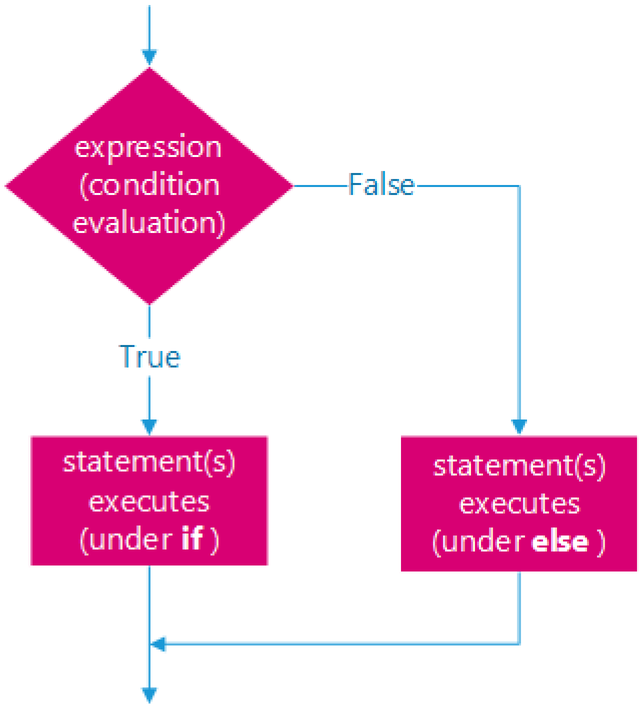
Python works in the same way. Here's an example:

In [ ]:
```
raining = True
if raining:
    print("I will not go to the store.")
else:
    print("It looks sunny outside. I'm going to the store.")
```

What happened here?
In the first line, we defined a boolean variable called `raining` . Remember our friends, boolean variables? These are variables that are either `True` or `False` . They can't be anything else!
What happens if we run the same exact code, but change `raining` to be `False` ?

In [ ]:
```
raining = False
if raining:
    print("I will not go to the store.")
else:
    print("It looks sunny outside. I'm going to the store.")
```

A **conditional** statement is essentially a flowchart diagram.


expression (condition evaluation)
False
True
statement(s) executes (under **if**)
statement(s) executes (under **else**)

(Image courtesy of electricalworkbook.com)

We can add some more arrows and conditions to our flowchart. What if the weather is cloudy, foggy, or there is a tornado watch?

In [ ]:
```python
raining = False
foggy = False
tornado_watch = False

if raining:
    print("I will not go to the store.")
elif foggy:
    print("I can't see a single thing out there.")
elif tornado_watch:
    print("Gadzooks!")
else:
    print("It looks sunny outside. I'm going to the store.")
```

Note that `elif` is shorthand for "else, if". Python will check the `if` statement first. Remember that there are two possibilities for this `if` statement:

- In cases where the `if` statement is `True` , Python will go on to execute the indented code that immediately follows it.
- In cases where the `if` statement is `False` , Python will go on to check the `elif` and `else` statements that follow it. In cases where none of the `if` and `elif` statements or true, Python will execute the `else` statement.

An `if` statement always needs an `else` , in case the `if` statement is not `True` !

Try changing the value of each of the boolean variables, and see what happens.
Notice that each conditional statement is followed by an **indented line**. In Python, indentations are very important. The indented lines "belong" to the unindented conditional statements that are right above it. Try removing indentations or indenting too much--you will receive some nasty-looking error messages!

When indenting, you can either use 4 spaces or 1 'Tab'. Whatever you choose, it's important to keep it consistent--use one or the other, but not both.
Also note that the first line of each "block" must end in a colon (**:**).

Unsurprisingly, we can use math operators and symbols as our conditional statements. To **compare** the values of two different variables or numbers, we will need to use **double equals signs** (==). Remember that one equals sign (=) will make Python think that you are trying to assign a variable!

Read the following code. What will it print out? Check your guesses by running the cell.

In [ ]:
```
if 5 > 3:
    print("5 is greater than 3. I knew that!")
else:
    print("5 is not greater than 3. Your math is wrong.")

if 4==2*2:   # Note that we have to use == when comparing values. One = means
 that we are trying to define a variable!
    print("4 is equal to 2 * 2.")
else:
    print("4 is not the square of 2!")

if "tomato"=="tomato":
    print("to-may-to, to-mah-to")
else:
    print("Nonsense!")
```

Is this what you expected? Explain your reasoning below.

Answer Here

Part 1: For Loops

For most things in programming, we want to maximize efficiency. Let's say I have a list of numbers from 1-10, and I want to multiply each element in the list by 2. Here's an example of code I could write:

In [ ]:
```
numbers_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numbers_list[0] = numbers_list[0] * 2
numbers_list[1] = numbers_list[1] * 2
numbers_list[2] = numbers_list[2] * 2
numbers_list[3] = numbers_list[3] * 2
```

Oof! I've only changed four of the numbers, and I'm already tired of typing! Is there a better way we can do this? *Of course there is!* Instead of changing each element within a list at once, we can **iterate** over the entire list. In programming, **iterating over** a collection of items means that we are going to apply an operation to each item in the collection. In our case, we want to **iterate over each element** in this list. We can do this with a **for loop**.

In [ ]:
```
# Run this code!
numbers_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for N in numbers_list:
    print(N, "times 2 is", N*2)
```

Much better! In the previous cell, the variable `N` is a placeholder name for each element in the list. We used this placeholder name to refer to elements. Python realizes that we want to iterate over each element within the list called `numbers_list`.

After the `for` statement, Python will look to see if there is indented code after it. Indeed, there is! We say that this indented code is **inside** the loop.

Python will apply the code inside the loop to every single element in the list, until it reaches the last one.

The following code does the same thing. However, it iterates over each **index** in the list, rather than each **element** of the list. This way, we can also update the original list one element at a time. We'll print out this updated list at the end, just to see what changed.

In [ ]:
```
# Run this code!
numbers_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for i in range(len(numbers_list)):
    print(numbers_list[i], "times 2 is", numbers_list[i]*2)
    numbers_list[i] = numbers_list[i] * 2 # Update each element of the list

print("Updated numbers:", numbers_list)
```

Using the `for i in range(len(list_variable))` method is a very common way of iterating over lists. Here, `i` was the placeholder variable that referred to all of the possible positions/indices within a list.

You can also iterate over characters within a string! What do you think the following code does?

In [ ]:
```
for letter in 'Hello!':
    print(letter)
```

Your Turn!

Believe it or not, but you now have the tools to code up your own reverse transcriptase! Here is your task:

1. I'll give you a short RNA sequence. You'll have to set up a `for` loop that iterates over each letter in the sequence.
2. Inside of the `for` loop, you'll have to write `if`, `elif`, and `else` statements. What should the reverse transcriptase give back if it sees an 'A'? If it's not an 'A', but a 'G' instead? and so on.
3. Concatenate each of these letters onto the string called `result`. Go back to Tutorial 0 if you are unfamiliar with string concatenation.
4. Finally, print out the `result` of your reverse transcriptase. Don't worry about splicing, etc.

In [2]:
```
rna_seq = 'UUCAUGC'
result = ''

# The following line may come in handy
result = result + '*'

# Your Code After This Line
```

In [ ]:
```
```