Capitalization, Indentation, String Splicing

1. **Capitalization is important in Python**

   There is a difference between `True` and `true`. Each programming language is different. In Python, `True` and `False` are the values of the Boolean type.
   Note, a Boolean type is one of the most primitive data types, with the logical values of True or False.

   Further, each programming language has a set of keywords, which are special words that are used by the language itself. These are also case-sensitive. You cannot use these keywords as variable names. Python 2.7 has 31 keywords. They are:

`and`	`del`	`from`	`not`	`while`
`as`	`elif`	`global`	`or`	`with`
`assert`	`else`	`if`	`pass`	`yield`
`break`	`except`	`import`	`print`	
`class`	`exec`	`in`	`raise`	
`continue`	`finally`	`is`	`return`	
`def`	`for`	`lambda`	`try`	

2. **Indentation is important in Python**

   Notice that these notes are easy to read with indentation. Same goes for your code!

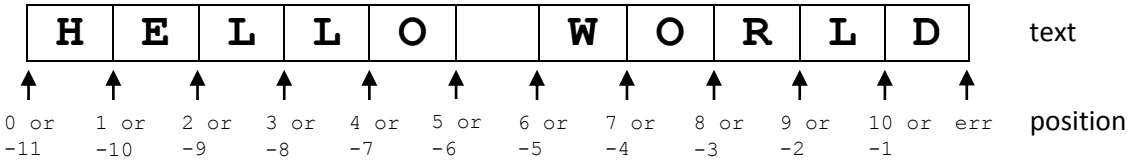
   When you write Python code, make sure to indent where necessary. You indent the line after a line that has a colon at the end (`:`). The following lines in your code stay indented to represent what you want to execute inside that code block. Once you have finished writing your code block with indentation, you decrease the indent to go back to the normal program flow.

```
if some_condition:
    INDENTED CODE BLOCK

elif some_other_condition:
    INDENTED CODE BLOCK

else:
    INDENTED CODE BLOCK
```

```
for some_iteration:
    INDENTED CODE BLOCK

while some_condition_holds:
    INDENTED CODE BLOCK
```

3. **Indexing a string**

These can be a little confusing. I like to think of it like the following illustration.



The square brackets define the slicing operations to do on the string. They numbers inside the square brackets stand for, in this order:

```
text[ beginning : end : step ]
```

By default `step` is 1 (as in, we take do not skip any letters).
So we can ask the following types of questions:

a. What is the substring `text[3:9]` ? Now you can look at the arrows. Notice that we stop once we hit the arrow pointing to 9, and do not take the letter after it. This is also equivalent to `text[3:-2]` according to the position numbering. If we want to be explicit and also include the step, this is equivalent to `text[3,9,1]`.

   The result would be the string `LO WOR`

b. What is `text[3]`? This is equivalent to getting `text[3:4]` or `text[3:4:1]`.
   You can look at where the arrow for 3 points and take the next letter – in this case, `L`.

c. What about when we want to take every other letter, or every third letter, or go backwards? Now we can change the step. What is `text[1:9:3]`? Look at the arrows and take every 3rd letter between positions 1 to 8 (this is the substring ELLO WOR).

   The result is the string `EOO`.

   Similarly, to go backwards we can set the step to be negative. What is `text[-2:1:-2]`? This means we start at position -2 and go through the string backwards until position 1, taking every other letter.

   The result is the string `LO L` (note the space between O and L).

   Note. If we tried to take `text[1:9:-2]`, this means start at position 1 and go to position 9 backwards by every 2nd letter. This gives an empty string `''` because there is no string going from position 1 to 9 going backwards. To achieve what we were thinking of achieving, we would have to write `text[9:1:-2]`. This gives the string `LO L`.