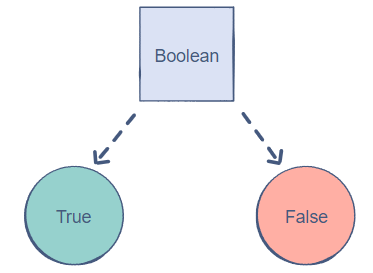
**Booleans**

This lesson highlights the key features of the Boolean data type.

The **Boolean** (also known as **bool**) data type allows us to choose between two values: *true* and *false*.



In Python, we can simply use True or False to represent a bool:

print(True)

f\_bool = False

print(f\_bool)

**Note**: The first letter of a bool needs to be capitalized in Python.

A Boolean is used to determine whether the logic of an expression or a comparison is correct. It plays a huge role in data comparisons.

# Strings

This lesson highlights the key features of the string data type.

**We'll cover the following**

* [The Length of a String](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6353203391365120#The-Length-of-a-String)
* [Indexing](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6353203391365120#Indexing)
* [Accessing Characters](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6353203391365120#Accessing-Characters)
  + [Reverse Indexing](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6353203391365120#Reverse-Indexing)
* [String Immutability](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6353203391365120#String-Immutability)
* [ASCII Versus Unicode](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6353203391365120#ASCII-Versus-Unicode)

A group of characters such as this is an example of the string data type.

A string is a collection of characters closed within single, double or triple quotation marks.

A string can also contain a single character or be entirely empty.

print("Harry Potter!")  *# Double quotation marks*

got = 'Game of Thrones...'  *# Single quotation marks*

print(got)

print("$")  *# Single character*

empty = ""

print(empty)  *# Just prints an empty line*

multiple\_lines = '''Triple quotes allows

multi-line string.'''

print(multiple\_lines)

From the examples above we can see:

* A blank space inside the string quotation marks is also considered to be a character.
* To add a multi-line string we can use triple quotes.

## The Length of a String

The length of a string can be found using the len() built-in function. This length indicates the number of characters in the string:

random\_string = "I am Batman"  *# 11 characters*

print(len(random\_string))

## Indexing

In a string, every character is given a numerical **index** based on its position.

A string in Python is indexed from 0 to n-1 where n is its length. This means that the index of the first character in a string is 0.

## Accessing Characters

Each character in a string can be accessed using its index. The index must be closed within square brackets, [], and appended to the string.

batman = "Bruce Wayne"

first = batman[0]  *# Accessing the first character*

print(first)

space = batman[5]  *# Accessing the empty space in the string*

print(space)

last = batman[len(batman) - 1]

print(last)

*# The following will produce an error since the index is out of bounds*

*# err = batman[len(batman)]*

If we try to execute the code on **line 12**, we would get an error because the maximum index is len(batman) - 1. A higher value is not within the bounds of the string. Since len(batman) is larger than len(batman) - 1, it will produce an error.

### Reverse Indexing

We can also change our indexing convention by using negative indices.

Negative indices start from the opposite end of the string. Hence, the -1 index corresponds to the last character:

batman = "Bruce Wayne"

print(batman[-12])  *# Corresponds to batman[10]*

print(batman[-11])  *# Corresponds to batman[6]*

## String Immutability

Once we assign a value to a string, we can’t update it later. How about verifying it with an executable below?

string = "Immutability"

string[0] = 'O' *# Will give error*

The above code gives TypeError because Python doesn’t support item assignment in case of strings.

Remember, assigning a new value to string variable doesn’t mean that you’ve changed the value. Let’s verify it with the id() method below.

str1 = "hello"

print(id(str1))

str1 = "bye"

print(id(str1))

Notice, when we assign a new value to str1 (at **line 4**) its identity changes not the value.

## ASCII Versus Unicode

In Python 3.x, all strings are unicode. But, older versions of Python (Python 2.x) support only ASCII characters. To use unicode in Python 2.x, preceding the string with a u is must. For example:

string = u"This is unicode"

# String Slicing

In this lesson, we'll understand what slicing is and how it can be applied to strings.

**We'll cover the following**

* [Definition](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/4859749259804672#Definition)
* [Slicing with a Step](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/4859749259804672#Slicing-with-a-Step)
* [Reverse Slicing](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/4859749259804672#Reverse-Slicing)
* [Partial Slicing](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/4859749259804672#Partial-Slicing)

## Definition

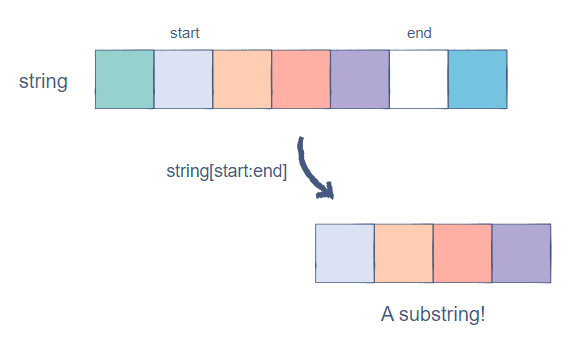
**Slicing** is the process of obtaining a portion (substring) of a string by using its indices.

Given a string, we can use the following template to slice it and obtain a substring:

string[start:end]

* start is the index from where we want the substring to start.
* end is the index where we want our substring to end.

The character at the end index in the string, will not be included in the substring obtained through this method.



my\_string = "This is MY string!"

print(len(my\_string))

print(my\_string[0:4]) # From the start till before the 4th index

print(my\_string[1:7])

print(my\_string[8:len(my\_string)]) # From the 8th index till the end

## Slicing with a Step

Python 3 also allows us to slice a string by defining a **step** through which we can skip characters in the string. The default step is 1, so we iterate through the string one character at a time.

The step is defined after the end index:

string[start:end:step]

Let’s see how this works:

my\_string = "This is MY string!"

print(my\_string[1:7]) # A step of 1

print(my\_string[0:7:3]) # A step of 2

print(my\_string[0:7:5]) # A step of 5

## Reverse Slicing

Strings can also be sliced to return a reversed substring. In this case, we would need to switch the order of the start and end indices.

A negative step must also be provided:

my\_string = "This is MY string!"

print(my\_string[5:0:-1]) # Take 1 step back each time

print(my\_string[17:0:-2]) # Take 2 steps back. The opposite of what happens in the slide above

## Partial Slicing

One thing to note is that specifying the start and end indices is **optional**.

If start is not provided, the substring will have all the characters until the end index.

If end is not provided, the substring will begin from the start index and go all the way to the end.

Let’s see this in action:

my\_string = "This is MY string!"

print(my\_string[:8]) # All the characters before 'M'

print(my\_string[8:]) # All the characters starting from 'M'

print(my\_string[:]) # The whole string

print(my\_string[::-1]) # The whole string in reverse (step is -1)

That’s pretty much all we need to know about string slicing. Play around with the strings above to get a better understanding of how slicing works.

# The if-elif-else Statement

This lesson highlights the main properties of the `if-elif-else` statement.

**We'll cover the following**

* [Structure](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6149440261062656#Structure)
* [Multiple elif Statements](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6149440261062656#Multiple-elif-Statements)

The if-else statement handles two sides of the same condition: True and False. This works very well if we’re working with a problem that only has two outcomes.

However, in programming, it isn’t always a True or False scenario, and a problem can have multiple outcomes.

This is where the if-elif-else statement shines. It is the most comprehensive conditional statement because it allows us to create multiple conditions easily.

The elif stands for **else if**, indicating that if the previous condition fails, try this one.

## Structure

The if and else blocks will remain the same. The elif statement comes in between the two.

Let’s write an if-elif-else statement which checks the state of a traffic signal and generates the appropriate response:

light = "Red"

if light == "Green":

print("Go")

elif light == "Yellow":

print("Caution")

elif light == "Red":

print("Stop")

else:

print("Incorrect light signal")

Now, our conditional statement caters to **all** possible values of light.

Try changing the value and see how the response changes.

## Multiple elif Statements

This is the beauty of the if-elif-else statement. We can have as many elifs as we require, as long as they come between if and else.

**Note**: An if-elif statement can exist on its own without an else block at the end. However, an elif cannot exist without an if statement preceding it (which naturally makes sense).

Let’s write a piece of code that checks whether the value of an integer is in the range of 0-9 and prints the word in English:

num = 5

if num == 0:

print("Zero")

elif num == 1:

print("One")

elif num == 2:

print("Two")

elif num == 3:

print("Three")

elif num == 4:

print("Four")

elif num == 5:

print("Five")

elif num == 6:

print("Six")

elif num == 7:

print("Seven")

elif num == 8:

print("Eight")

elif num == 9:

print("Nine")

An important thing to keep in mind is that an if-elif-else or if-elif statement is not the same as multiple if statements. if statements act **independently**.

If the conditions of two successive ifs are True, both statements will be executed.

On the other hand, in if-elif-else, when a condition evaluates to True, the rest of the statement’s conditions are not evaluated.

We’ll understand this better through an example:

As we can see, in the if tab, all the statements are computed one by one. Hence, we get multiple outputs.

In the if-elif-else tab, since the first condition holds true, all the others are discarded. This proves to be more efficient in terms of code performance.

What would be the value of result at the end of this code?

num1 = 10  
num2 = 20  
result = 4000  
  
if (num1 > 50 or not num2 <= 5):  
    result = num1 \* num2  
else:  
    result = num1 + num2

**A)**

200

**Explanation**

The if condition is fulfilled because not num2 <=5 is true. num2 is in fact greater than 5. Hence, num1 and num2 are multiplied.

# Exercise: Discounted Price

Let's make a price discounting system!

**We'll cover the following**

* [Problem Statement](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6440310629728256#Problem-Statement)
  + [Sample Input](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6440310629728256#Sample-Input)
  + [Sample Output](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6440310629728256#Sample-Output)
* [Coding Challenge](https://www.educative.io/module/page/An5VrvSGJoxKE7zz5/10370001/6340021872492544/6440310629728256#Coding-Challenge)

## Problem Statement

In this challenge, you must discount a price according to its value.

* If the price is 300 or above, there will be a 30% discount.
* If the price is between 200 and 300 (200 inclusive), there will be a 20% discount.
* If the price is between 100 and 200 (100 inclusive), there will be a 10% discount.
* If the price is less than 100, there will be a 5% discount.
* If the price is negative, there will be no discount.