

MY JOURNAL TO PYTHON

4110E211 尤拉梅

MY DEAR GREAT TEACHER



AGENDA

- Python
- Input and Output : input() and print
- Data Types : numeric, string, list, dict.
- Operation ON data type:
- CONTROLS : IF- | IF –ELSE IF | -F-ELSE
- LOOP : FOR |WHILE | RANGE () |BREAK | CONTINUE
- FUNCTION
 - ① PARAMETERS (ARGUMENTS)
 - ② RECURSIVE FUNCTION
 - ③ LAMBA FUNCTION

WHAT IS PYTHON?

- Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.
- It is used for:
- web development (server-side),
- software development,
- mathematics,
- system scripting.
- What can Python do?
- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.
- Why Python?
- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way

GOOD TO KNOW:

- The most recent major version of Python is Python 3, which we shall be using in this tutorial.
- However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- In this tutorial Python will be written in a text editor.
- It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Example



The screenshot shows a code editor interface. At the top, there are two tabs: '+ 程式碼' (Code) and '+ 文字' (Text). The 'Code' tab is active. Below the tabs, there is a code editor area. On the left side of the editor, there is a green checkmark and a timer showing '0 秒' (0 seconds). The code being executed is `print("Hello, Fighter!")`. Below the code, the output is displayed as 'Hello, Fighter!'.

```
+ 程式碼 + 文字
```

```
✓ 0 秒 print("Hello, Fighter!")
```

```
Hello, Fighter!
```

PYTHON SYNTAX



PYTHON SYNTAX CAN BE EXECUTED BY WRITING DIRECTLY IN THE COMMAND LINE:

```
>>> print("Hello, World!")  
Hello, World!
```

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

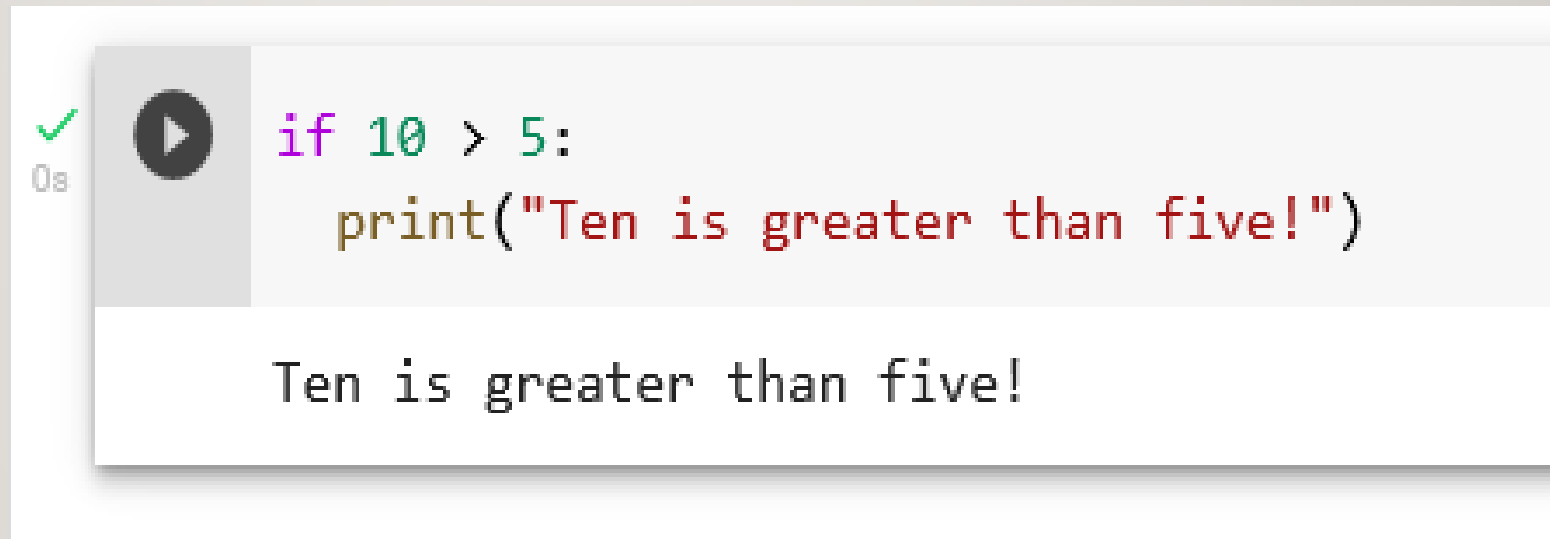
```
C:\Users\Your Name>python myfile.py
```

PYTHON INDENTATION

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

PYTHON USES INDENTATION TO INDICATE A BLOCK OF CODE.

- Example

A screenshot of a Python code execution environment. On the left, there is a green checkmark and the text '0s'. Next to it is a play button icon. The code being executed is:

```
if 10 > 5:  
    print("Ten is greater than five!")
```

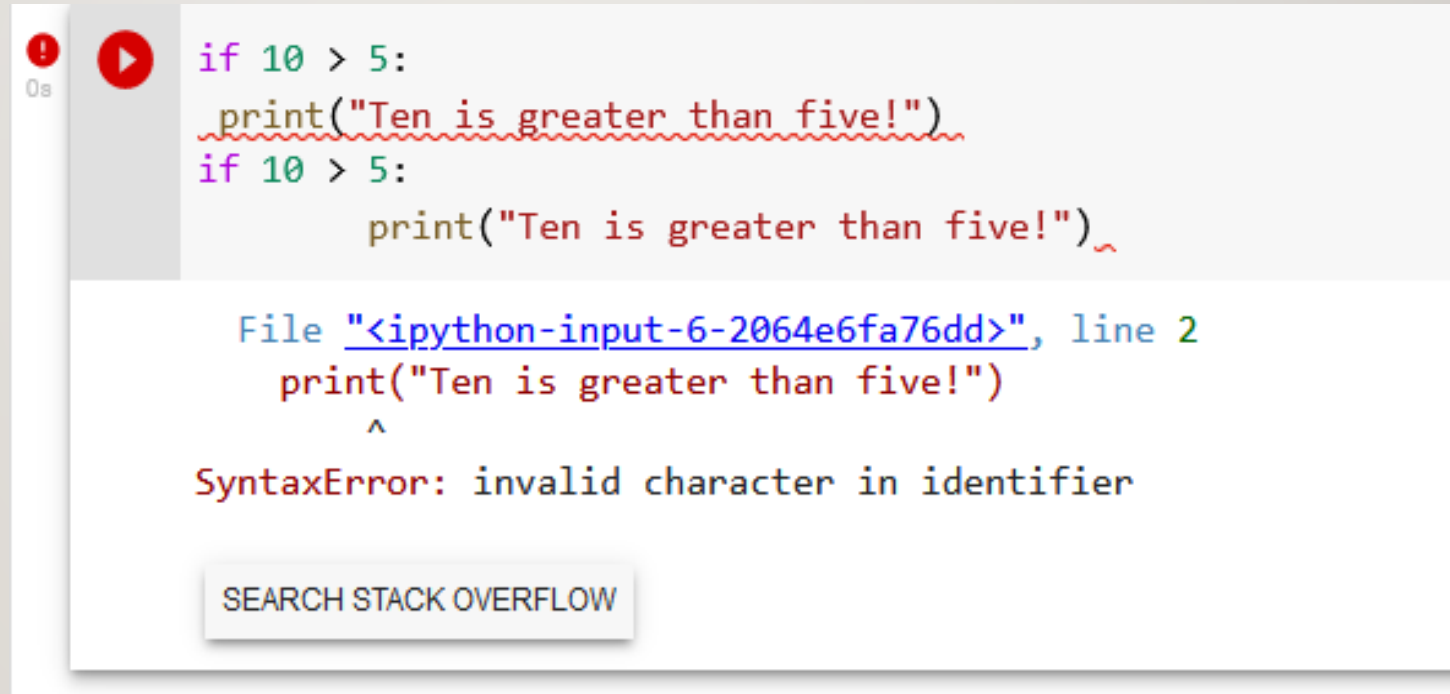
 Below the code, the output is displayed:

```
Ten is greater than five!
```

```
✓  
0s  
▶ if 10 > 5:  
    print("Ten is greater than five!")  
  
Ten is greater than five!
```


PYTHON WILL GIVE YOU AN ERROR IF YOU SKIP THE INDENTATION:

- **Example**



The screenshot shows a Jupyter Notebook interface. At the top left, there is a red circle with a white exclamation mark and the text '0s'. To its right is a red play button icon. The code cell contains the following Python code:

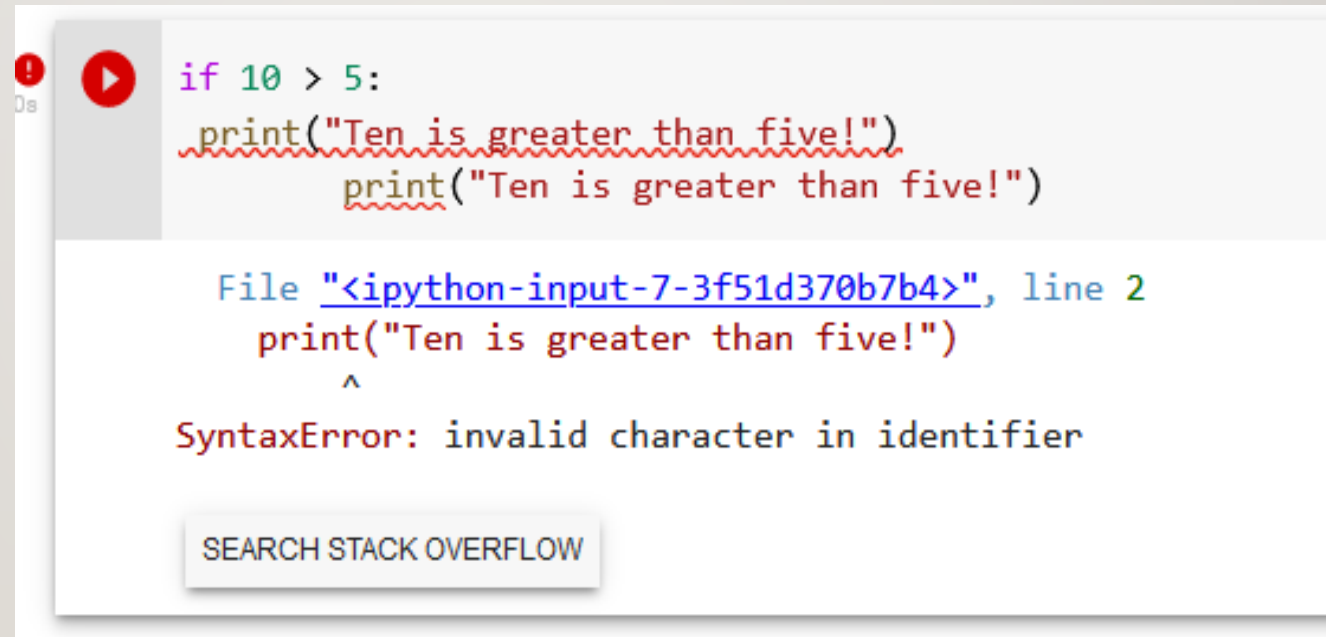
```
if 10 > 5:  
    print("Ten is greater than five!")  
if 10 > 5:  
    print("Ten is greater than five!")
```

The second `print` statement is underlined with a red wavy line, indicating an error. Below the code, the error message is displayed:

```
File "<ipython-input-6-2064e6fa76dd>", line 2  
    print("Ten is greater than five!")  
    ^  
SyntaxError: invalid character in identifier
```

At the bottom of the cell, there is a button labeled "SEARCH STACK OVERFLOW".

Example

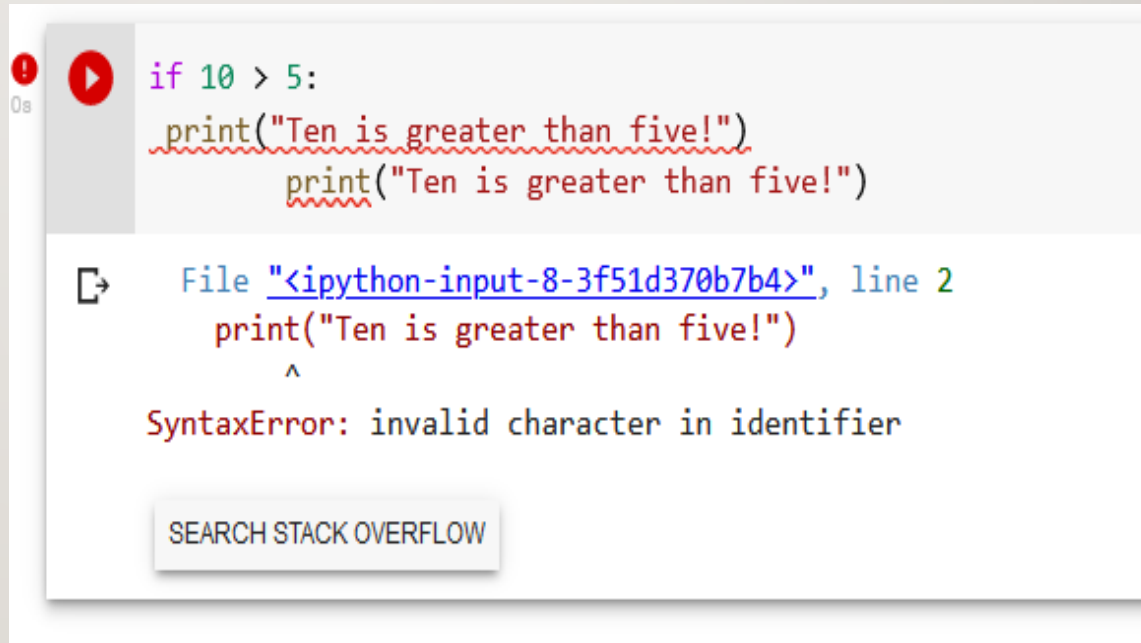
A screenshot of a Jupyter Notebook interface showing a syntax error. The code cell contains two lines of Python code: `if 10 > 5:` followed by `print("Ten is greater than five!")` on the next line. The second line is underlined with a red wavy line, indicating an error. Below the code, the error message is displayed: `File "<ipython-input-7-3f51d370b7b4>", line 2` followed by `print("Ten is greater than five!")` and a caret (^) pointing to the opening quote of the string. The error message is `SyntaxError: invalid character in identifier`. At the bottom of the error message box is a button that says "SEARCH STACK OVERFLOW".

```
if 10 > 5:  
    print("Ten is greater than five!")  
    print("Ten is greater than five!")
```

File "<ipython-input-7-3f51d370b7b4>", line 2
 print("Ten is greater than five!")
 ^
SyntaxError: invalid character in identifier

SEARCH STACK OVERFLOW

YOU HAVE TO USE THE SAME NUMBER OF SPACES IN THE SAME BLOCK OF CODE, OTHERWISE PYTHON WILL GIVE YOU AN ERROR:

A screenshot of a Jupyter Notebook interface showing a syntax error. The code cell contains two lines of Python code: `if 10 > 5:` followed by `print("Ten is greater than five!")` on the next line. The second line is indented with two spaces. The error message below the code states: `File "<ipython-input-8-3f51d370b7b4>", line 2` followed by `print("Ten is greater than five!")` with a caret under the first space, and `SyntaxError: invalid character in identifier`. A "SEARCH STACK OVERFLOW" button is visible at the bottom.

```
if 10 > 5:
    print("Ten is greater than five!")
    print("Ten is greater than five!")
```

File "<ipython-input-8-3f51d370b7b4>", line 2
 print("Ten is greater than five!")
 ^
SyntaxError: invalid character in identifier

SEARCH STACK OVERFLOW

IN PYTHON, VARIABLES ARE CREATED WHEN YOU ASSIGN A VALUE TO IT:

- **Example**

A code editor snippet with a light gray background. On the left, there is a green checkmark and the text '0s'. To the right of this is a dark gray play button icon. The code itself is displayed in a white box on the right, with 'x = 5' in green and 'y = "Hello, World!"' in red.

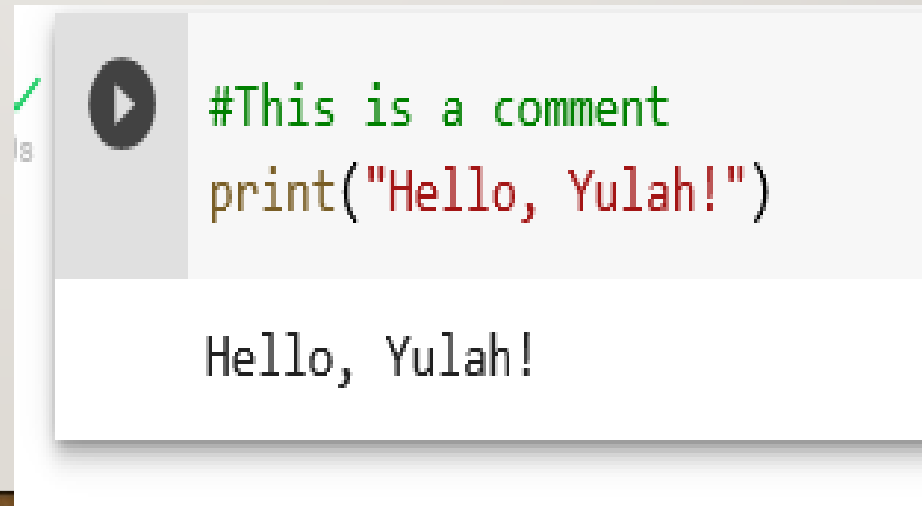
```
x = 5  
y = "Hello, World!"
```

PYTHON HAS NO COMMAND FOR DECLARING A VARIABLE.

COMMENTS

- Python has commenting capability for the purpose of in-code documentation.
- Comments start with a `#`, and Python will render the rest of the line as a comment:

Example



```
#This is a comment  
print("Hello, Yulah!")
```

Hello, Yulah!

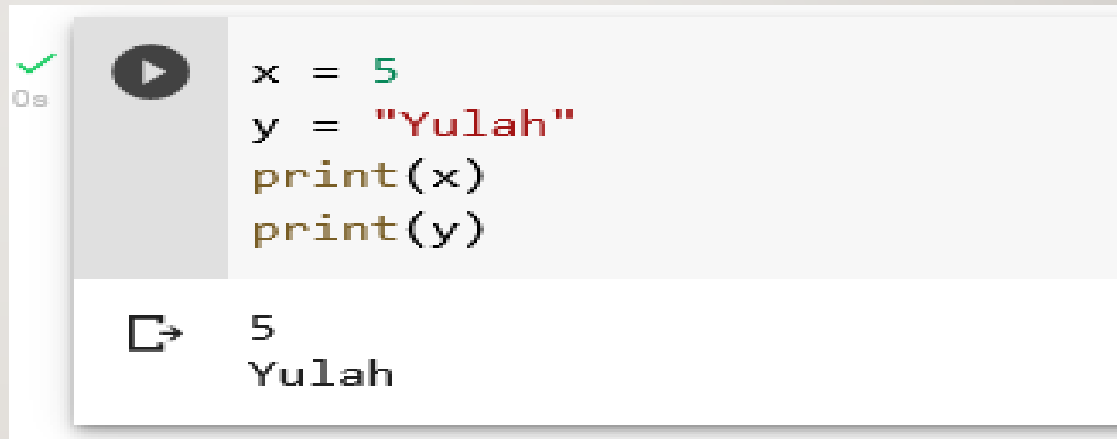
The image shows a code execution window. On the left, there is a green checkmark and a play button icon. The code is displayed in a light gray box with syntax highlighting: the comment is green, and the print statement is black. Below the code, the output "Hello, Yulah!" is shown in a white box.

PYTHON VARIABLES

- In Python, variables are created when you assign a value to it:

EXAMPLE

Variables in Python:

A screenshot of a Python code execution environment. On the left, there is a green checkmark and the text '0s'. In the center, there is a play button icon. To the right of the play button, the following Python code is displayed:

```
x = 5
y = "Yulah"
print(x)
print(y)
```

 Below the code, there is a copy icon (two overlapping squares) and the output of the code:

```
5
Yulah
```

```
x = 5
y = "Yulah"
print(x)
print(y)
```

```
5
Yulah
```


- Python has no command for declaring a variable.

You will learn more about variables in the Python Variables chapter.

COMMENTS

- Python has commenting capability for the purpose of in-code documentation.
- Comments start with a #, and Python will render the rest of the line as a comment:

Example



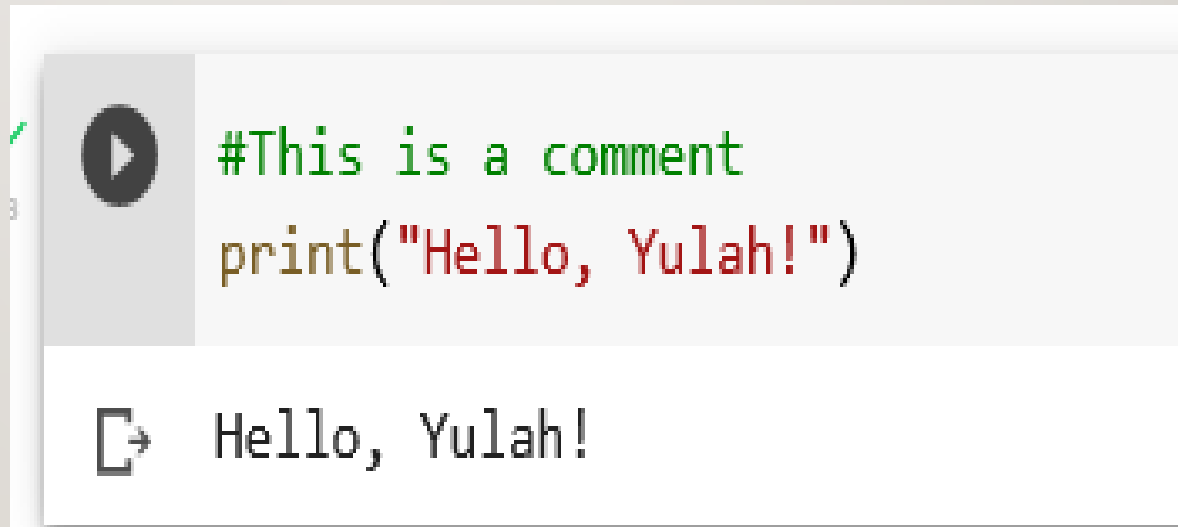
```
/ ▶ #This is a comment  
3 print("Hello, Yulah!")  
  
[ ] Hello, Yulah!
```

The image shows a code editor window with a light gray header. On the left, there is a vertical scrollbar and a line number '3'. The code is written in a monospaced font. The first line is a comment, '#This is a comment', in green. The second line is a print statement, 'print("Hello, Yulah!")', in black. Below the code, there is a white box with a gray border containing the output 'Hello, Yulah!' in black. To the left of the output is a small icon of a document with a right-pointing arrow.

Comments can be used to explain Python code.
Comments can be used to make the code more readable.
Comments can be used to prevent execution when testing code.

- **Creating a Comment**

Comments starts with a #, and Python will ignore them:

A screenshot of a code editor or terminal window. It shows a Python script with a comment and a print statement. The comment is highlighted in green, and the print statement is in a monospace font. Below the code, the output of the script is shown, indicating that the comment was ignored and the print statement executed successfully.

```
#This is a comment  
print("Hello, Yulah!")
```

↳ Hello, Yulah!

PYTHON COMMENTS

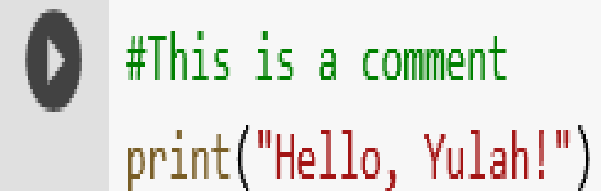


- Comments can be used to explain Python code.
-

- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

A code editor snippet with a light gray background. On the left, there is a dark gray vertical bar containing a white play button icon. To the right of this bar, the text "#This is a comment" is written in green, and "print('Hello, Yulah!')" is written in red. The text is monospaced and has a slight shadow.

```
#This is a comment  
print("Hello, Yulah!")
```

A terminal window snippet with a white background. It features a dark gray icon on the left that looks like a terminal window with a cursor. To the right of the icon, the text "Hello, Yulah!" is displayed in a monospaced font.

```
➤ Hello, Yulah!
```

COMMENTS CAN BE PLACED AT THE END OF A LINE, AND PYTHON WILL IGNORE THE REST OF THE LINE:

- **Example**



A screenshot of a Python code editor or interpreter window. The top bar is light gray and contains a play button icon on the left and a line number '8' on the right. The main area is white and displays the code `print("Hello, Yulah!") #This is a comment`. The word `print` is underlined with a red squiggly line. Below the code, there is a white box with a gray border containing the output `Hello, Yulah!` preceded by a small icon of a document with an arrow pointing out.

```
print("Hello, Yulah!") #This is a comment
```

```
Hello, Yulah!
```

A COMMENT DOES NOT HAVE TO BE TEXT THAT EXPLAINS THE CODE, IT CAN ALSO BE USED TO PREVENT PYTHON FROM EXECUTING CODE:



The image shows a Python code execution interface. On the left, there is a green checkmark and the text '0s'. To the right of this is a play button icon. The code being executed is: `#print("Hello, Yulah!")` and `print("Cheers, victory!")`. The output of the code is `Cheers, victory!`.

```
#print("Hello, Yulah!")  
print("Cheers, victory!")
```

Cheers, victory!

MULTI LINE COMMENTS

Python does not really have a syntax for multi line comments.
To add a multiline comment you could insert a `#` for each line:

Example



The screenshot shows a code editor window. On the left, there is a green checkmark and a play button icon. The code in the editor is as follows:

```
#This is a comment  
#written in  
#more than just one line  
print("Hello, Yulah!")
```

Below the code editor, there is a white box containing the output of the script:

```
→ Hello, Yulah!
```

OR, NOT QUITE AS INTENDED, YOU CAN USE A MULTILINE STRING.

- Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.



```
0s  """  
This is a comment  
written in  
more than just one line  
"""  
print("Hello, Yulah!")
```

 Hello, Yulah!

PYTHON VARIABLES

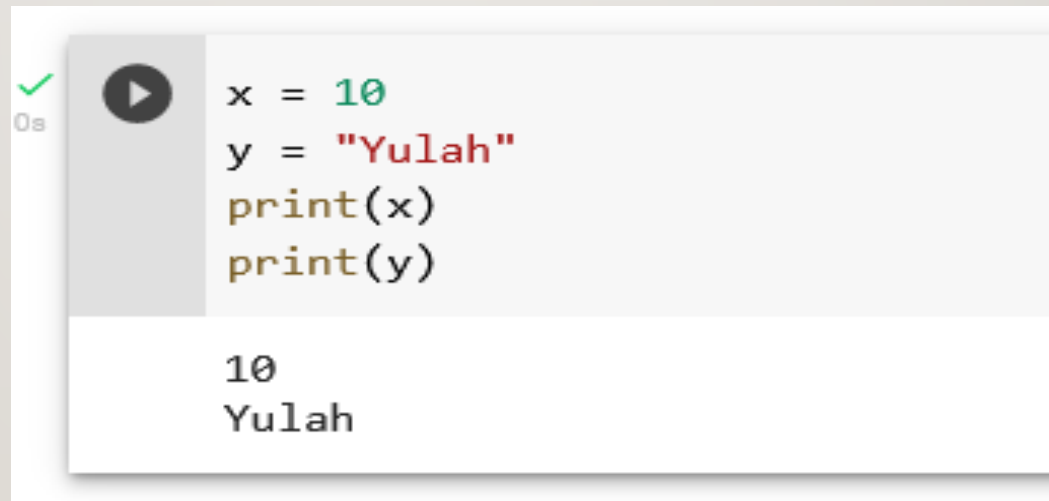
- **Variables**
- Variables are containers for storing data values.

CREATING VARIABLES

- Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

- **Example**

A screenshot of a Python code execution environment. On the left, there is a green checkmark and the text '0s'. Next to it is a play button icon. The code is as follows:

```
x = 10
y = "Yulah"
print(x)
print(y)
```

Below the code, the output is displayed:

```
10
Yulah
```

The code defines two variables, `x` and `y`. `x` is assigned the value `10` and `y` is assigned the string value `"Yulah"`. The `print` statements output the values of these variables.

VARIABLES DO NOT NEED TO BE DECLARED WITH ANY PARTICULAR TYPE, AND CAN EVEN CHANGE TYPE AFTER THEY HAVE BEEN SET.

- **Example**

✓
0s



```
x = 10      # x is of type int
x = "Yulah" # x is now of type str
print(x)
```

Yulah

CASTING

- If you want to specify the data type of a variable, this can be done with casting.

Example



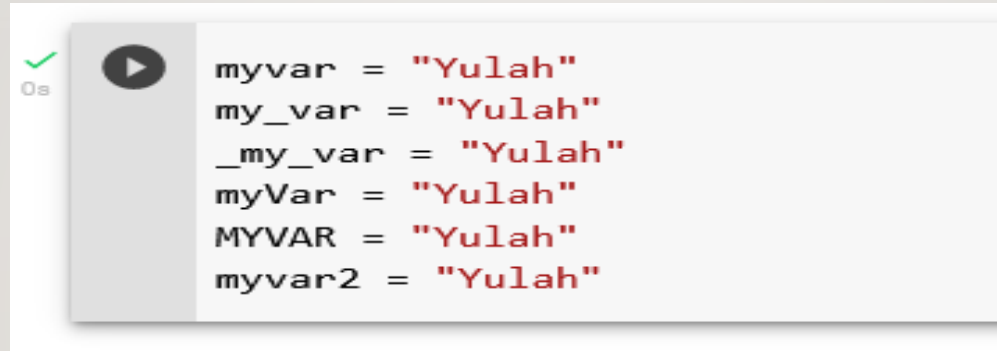
```
x = str(10)    # x will be '10'  
y = int(10)    # y will be 10  
z = float(10)  # z will be 10.0
```

PYTHON - VARIABLE NAMES

- **Variable Names**

- A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables: A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

EXAMPLE

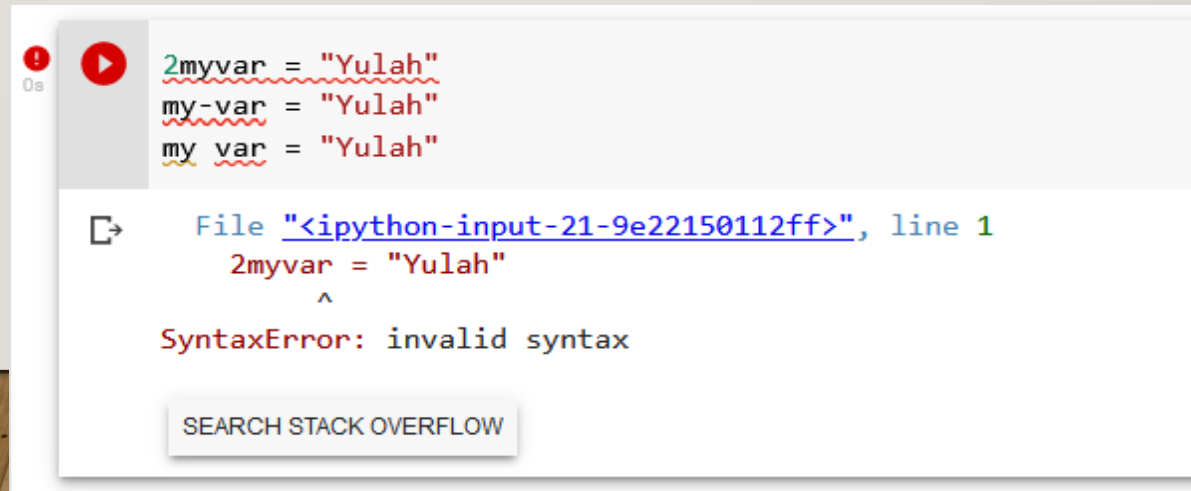


A screenshot of a Python code execution interface. On the left, there is a green checkmark icon and a play button icon. The code is as follows:

```
myvar = "Yulah"  
my_var = "Yulah"  
_my_var = "Yulah"  
myVar = "Yulah"  
MYVAR = "Yulah"  
myvar2 = "Yulah"
```

Example

Illegal variable names:



A screenshot of a Python code execution interface. On the left, there is a red exclamation mark icon and a play button icon. The code is as follows:

```
2myvar = "Yulah"  
my-var = "Yulah"  
my var = "Yulah"
```

Below the code, there is an error message:

```
File "<ipython-input-21-9e22150112ff>", line 1  
    2myvar = "Yulah"  
      ^  
SyntaxError: invalid syntax
```

At the bottom, there is a button labeled "SEARCH STACK OVERFLOW".

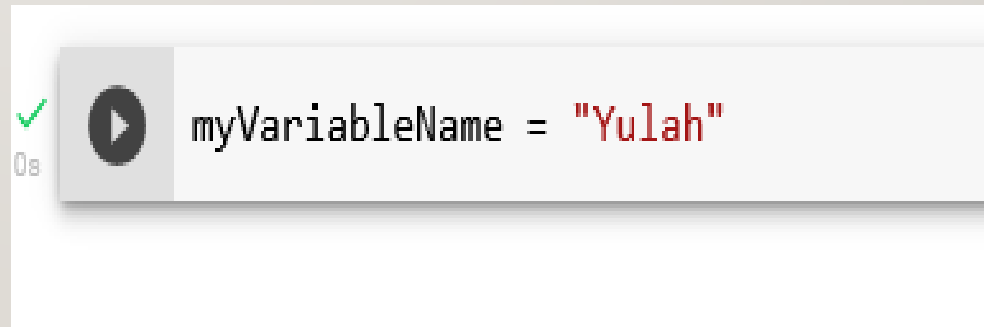
MULTI WORDS VARIABLE NAMES

- Variable names with more than one word can be difficult to read.
- There are several techniques you can use to make them more readable:

Camel Case

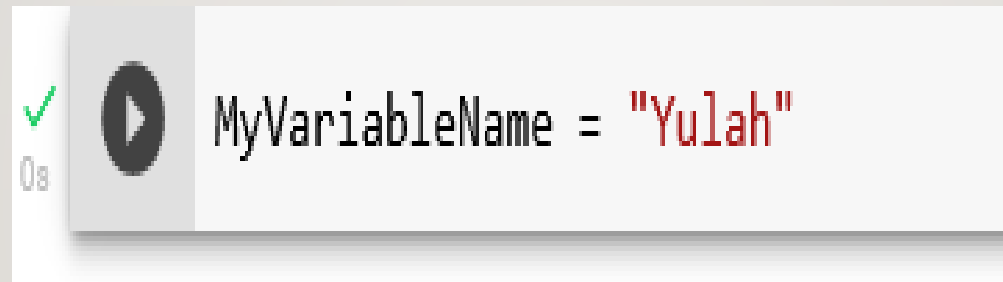
Each word, except the first, starts with a capital letter:

Each word starts with a capital letter:



PASCAL CASE

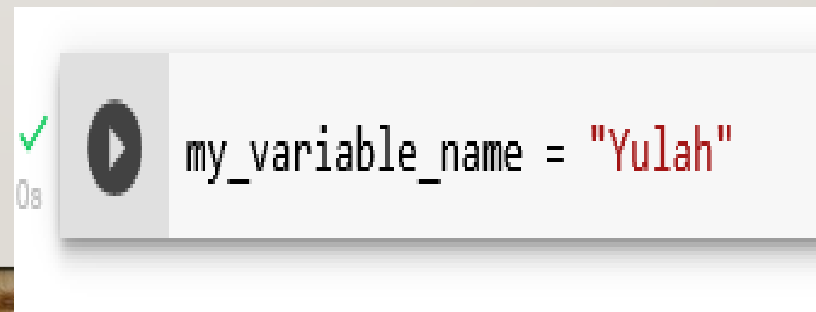
- Each word starts with a capital letter:



A code editor snippet showing a variable assignment in Pascal Case. On the left, there is a green checkmark and a play button icon. The code is `MyVariableName = "Yulah"`, where `MyVariableName` is in black and `"Yulah"` is in red.

Snake Case

Each word is separated by an underscore character:



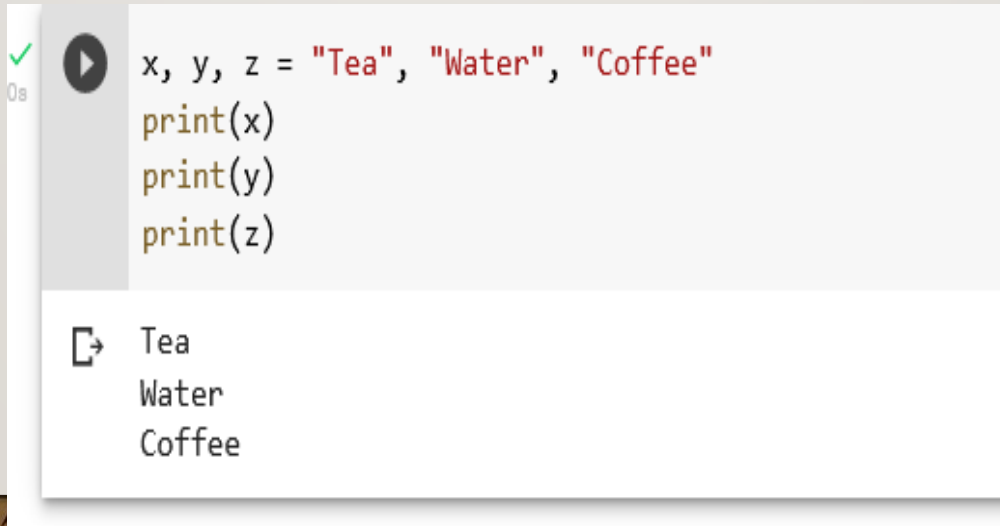
A code editor snippet showing a variable assignment in Snake Case. On the left, there is a green checkmark and a play button icon. The code is `my_variable_name = "Yulah"`, where `my_variable_name` is in black and `"Yulah"` is in red.

PYTHON VARIABLES - ASSIGN MULTIPLE VALUES

- **Many Values to Multiple Variables**

Python allows you to assign values to multiple variables in one line:

Example

A screenshot of a code editor showing a Python script being executed. The script assigns three string values to variables x, y, and z, and then prints each variable. The output shows the values 'Tea', 'Water', and 'Coffee' on separate lines. The background of the slide features a wooden floor texture at the bottom.

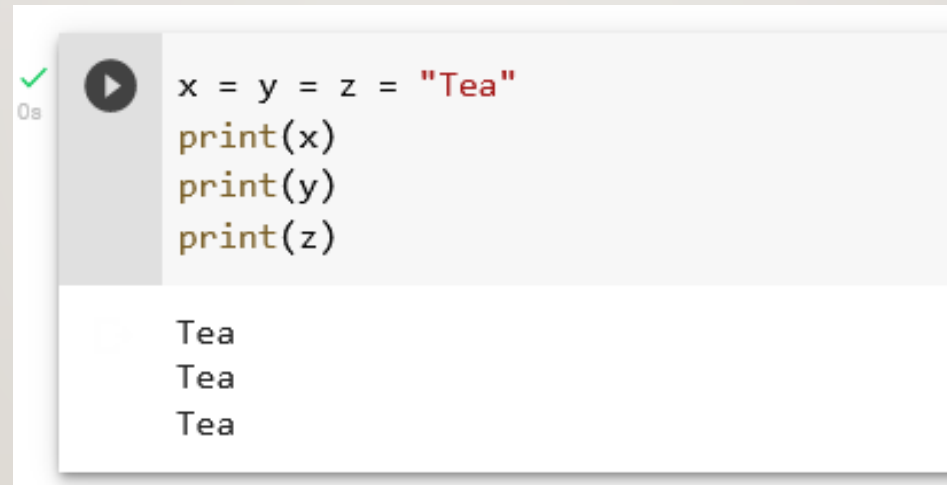
```
x, y, z = "Tea", "Water", "Coffee"
print(x)
print(y)
print(z)
```

Tea
Water
Coffee

ONE VALUE TO MULTIPLE VARIABLES

- And you can assign the *same* value to multiple variables in one line:

Example

A screenshot of a code editor or terminal window. On the left, there is a green checkmark and the text '0s'. To the right of this is a play button icon. The main area contains the following Python code:

```
x = y = z = "Tea"
print(x)
print(y)
print(z)
```

 Below the code, the output is displayed as three lines, each containing the word 'Tea'.

```
x = y = z = "Tea"
print(x)
print(y)
print(z)

Tea
Tea
Tea
```


UNPACK A COLLECTION

- If you have a collection of values in a list, tuple etc. Python allows you to extract the values into variables. This is called *unpacking*.

Example

```
drinks = ["tea", "water", "coffee"]
x, y, z = drinks
print(x)
print(y)
print(z)
```

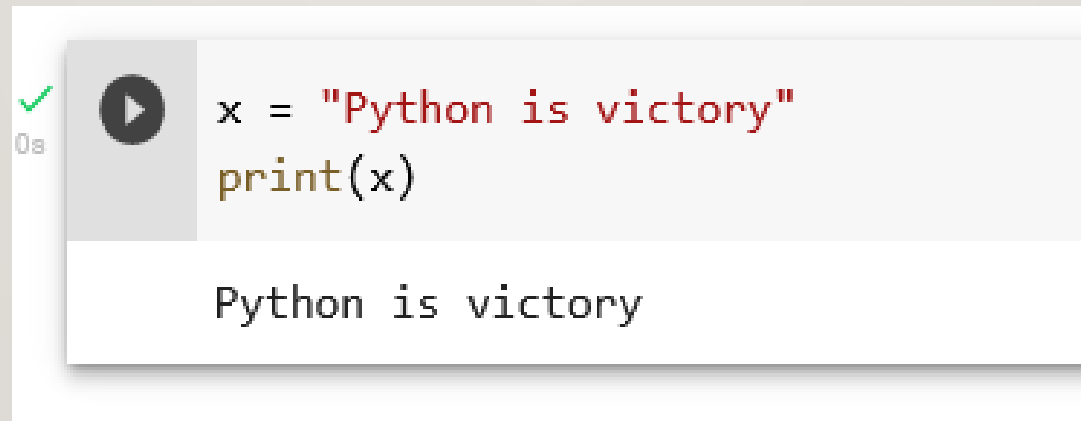
tea
water
coffee

PYTHON - OUTPUT VARIABLES

- **Output Variables**

The Python `print()` function is often used to output variables.

Example

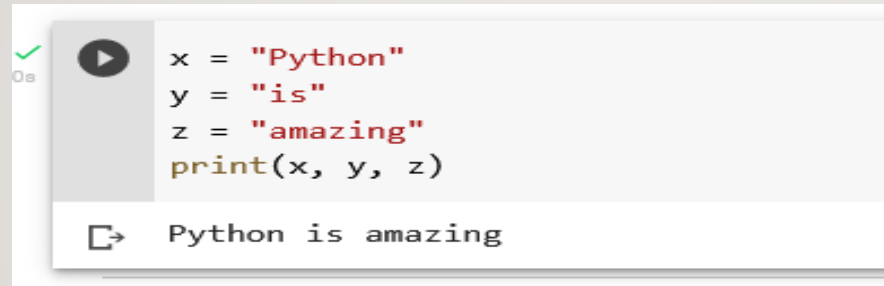
A screenshot of a code editor or terminal window. On the left, there is a green checkmark and the text '0s'. To the right of this is a play button icon. The main area contains two lines of Python code: `x = "Python is victory"` and `print(x)`. Below the code, the output 'Python is victory' is displayed.

```
x = "Python is victory"
print(x)
```

Python is victory

IN THE PRINT() FUNCTION, YOU OUTPUT MULTIPLE VARIABLES, SEPARATED BY A COMMA:

- **Example**



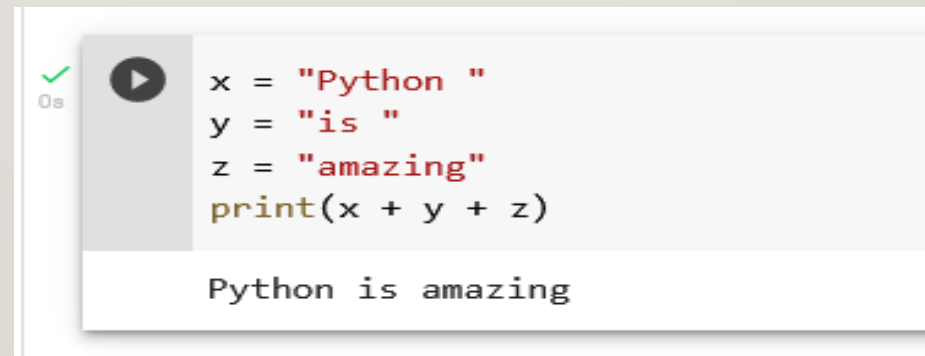
```
x = "Python"  
y = "is"  
z = "amazing"  
print(x, y, z)
```

Python is amazing

The image shows a code execution snippet with a green checkmark and '0s' in the top left corner. The code defines three variables: x = "Python", y = "is", and z = "amazing". It then uses the print function to output these variables separated by commas. The output displayed below the code is "Python is amazing".

You can also use the + operator to output multiple variables:

Example



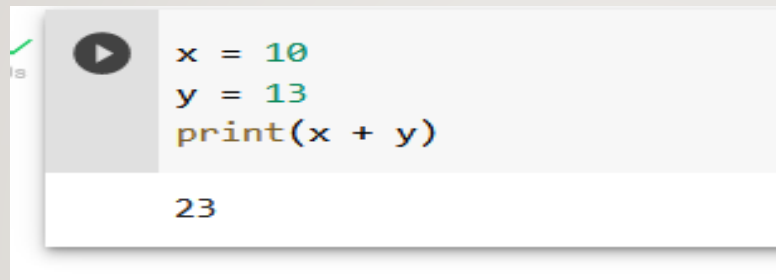
```
x = "Python "  
y = "is "  
z = "amazing"  
print(x + y + z)
```

Python is amazing

The image shows a code execution snippet with a green checkmark and '0s' in the top left corner. The code defines three variables: x = "Python ", y = "is ", and z = "amazing". It then uses the print function to output the concatenation of these variables using the + operator. The output displayed below the code is "Python is amazing".

FOR NUMBERS, THE + CHARACTER WORKS AS A MATHEMATICAL OPERATOR:

- **Example**

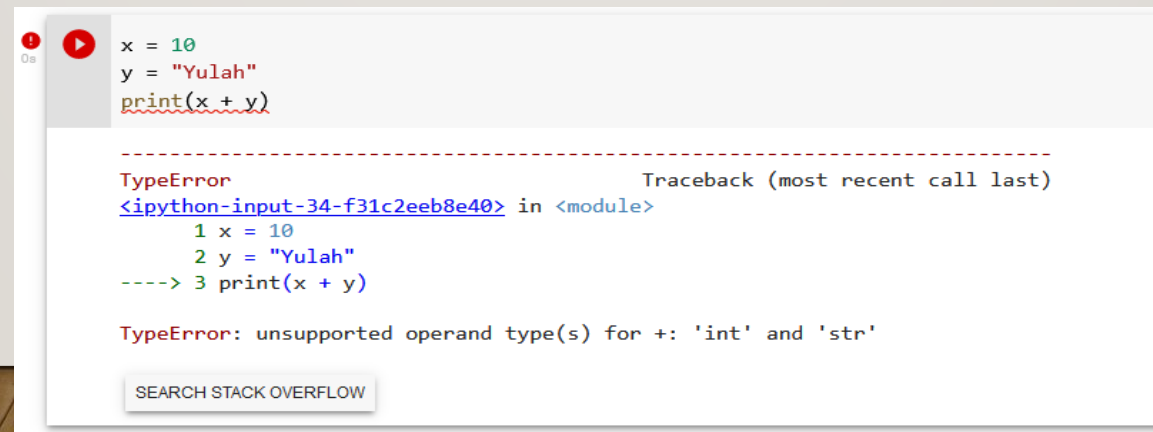


```
x = 10
y = 13
print(x + y)
```

23

In the print() function, when you try to combine a string and a number with the + operator, Python will give you an error:

Example



```
x = 10
y = "Yulah"
print(x + y)
```

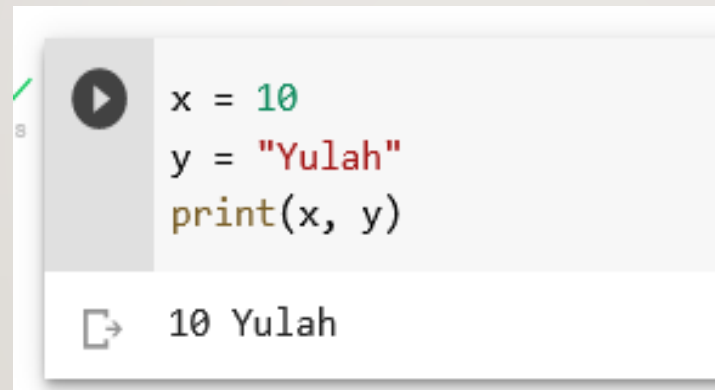
TypeError Traceback (most recent call last)
<ipython-input-34-f31c2eeb8e40> in <module>
 1 x = 10
 2 y = "Yulah"
----> 3 print(x + y)

TypeError: unsupported operand type(s) for +: 'int' and 'str'

SEARCH STACK OVERFLOW

THE BEST WAY TO OUTPUT MULTIPLE VARIABLES IN THE PRINT() FUNCTION IS TO SEPARATE THEM WITH COMMAS, WHICH EVEN SUPPORT DIFFERENT DATA TYPES:

- **Example**

A screenshot of a code editor showing a Python script being executed. The script defines two variables, x and y, and prints them. The output shows the values 10 and Yulah separated by a space.

```
x = 10  
y = "Yulah"  
print(x, y)
```

10 Yulah

PYTHON - GLOBAL VARIABLES

- **Global Variables**
- Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

Example



```
✓ 0s  x = "victory"

def myfunc():
    print("Python is " + x)

myfunc()

Python is victory
```

The image shows a code editor window with a green checkmark and '0s' in the top left corner. The code defines a global variable 'x' with the value 'victory', then defines a function 'myfunc()' that prints 'Python is ' followed by the value of 'x'. The function is then called, resulting in the output 'Python is victory'.

IF YOU CREATE A VARIABLE WITH THE SAME NAME INSIDE A FUNCTION, THIS VARIABLE WILL BE LOCAL, AND CAN ONLY BE USED INSIDE THE FUNCTION. THE GLOBAL VARIABLE WITH THE SAME NAME WILL REMAIN AS IT WAS, GLOBAL AND WITH THE ORIGINAL VALUE.

- **Example**

Create a variable inside a function, with the same name as the global variable



```
x = "victory"

def myfunc():
    x = "fantastic"
    print("Python is " + x)

myfunc()

print("Python is " + x)
```

Python is fantastic
Python is victory

The image shows a code execution interface. On the left, there is a green checkmark and a play button icon. The main area contains a Python script. The script defines a global variable 'x' with the value 'victory', then defines a function 'myfunc()' which creates a local variable 'x' with the value 'fantastic' and prints 'Python is fantastic'. After the function call, the script prints 'Python is victory' using the global variable 'x'. The output at the bottom shows the two lines of printed text.

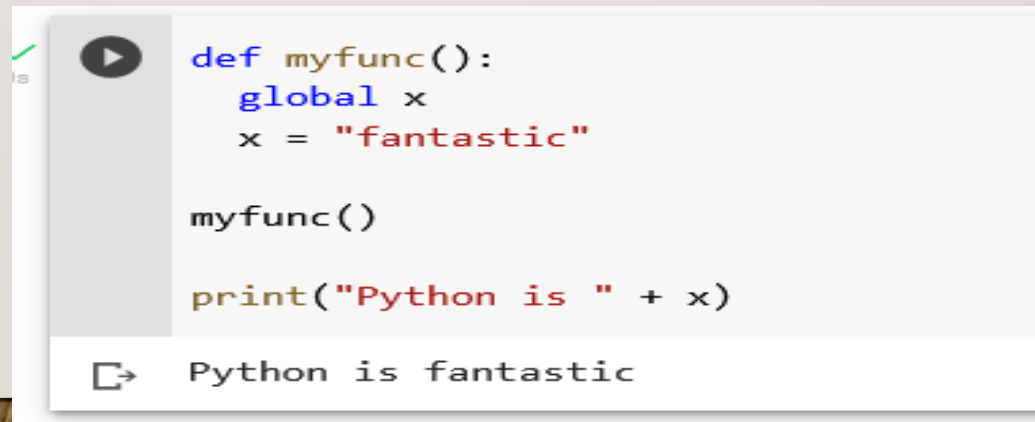
THE GLOBAL KEYWORD

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the global keyword.

Example

If you use the global keyword, the variable belongs to the global scope:




```
def myfunc():  
    global x  
    x = "fantastic"  
  
myfunc()  
  
print("Python is " + x)
```

Python is fantastic

ALSO, USE THE GLOBAL KEYWORD IF YOU WANT TO CHANGE A GLOBAL VARIABLE INSIDE A FUNCTION.

- **Example**
- To change the value of a global variable inside a function, refer to the variable by using the global keyword:



```
x = "victory"

def myfunc():
    global x
    x = "fantastic"

myfunc()

print("Python is " + x)
```

Python is fantastic

The image shows a code editor window with a green checkmark and a play button icon. The code defines a global variable `x` with the value "victory". A function `myfunc()` is defined that uses the `global` keyword to access and modify the global variable `x`, setting it to "fantastic". The function is then called, and the result is printed: "Python is fantastic".

PYTHON DATA TYPES



BUILT-IN DATA TYPES

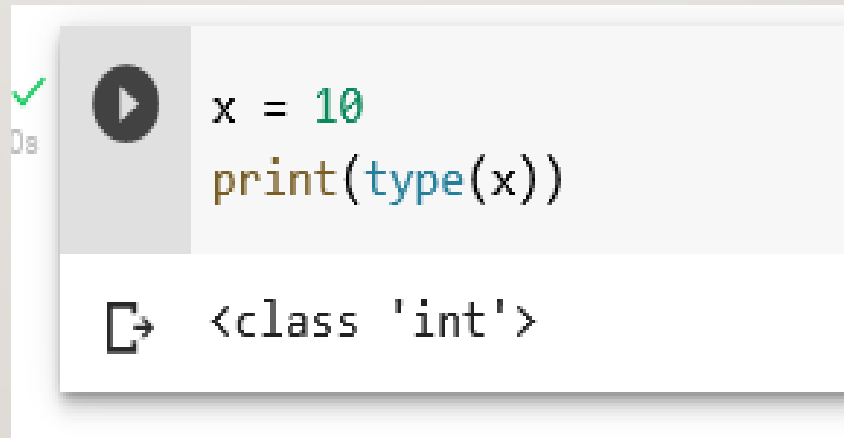
- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.

PYTHON HAS THE FOLLOWING DATA TYPES BUILT-IN BY DEFAULT, IN THESE CATEGORIES:

Text Type:	<code>str</code>
Numeric Types:	<code>int</code> , <code>float</code> , <code>complex</code>
Sequence Types:	<code>list</code> , <code>tuple</code> , <code>range</code>
Mapping Type:	<code>dict</code>
Set Types:	<code>set</code> , <code>frozenset</code>
Boolean Type:	<code>bool</code>
Binary Types:	<code>bytes</code> , <code>bytearray</code> , <code>memoryview</code>
None Type:	<code>NoneType</code>

GETTING THE DATA TYPE

- You can get the data type of any object by using the `type()` function:
- **Example**
- Print the data type of the variable `x`:



```
x = 10
print(type(x))

<class 'int'>
```

The image shows a code execution snippet. On the left, there is a green checkmark and a play button icon. The code consists of two lines: `x = 10` and `print(type(x))`. Below the code, the output is displayed as `<class 'int'>`.

SETTING THE DATA TYPE



IN PYTHON, THE DATA TYPE IS SET WHEN YOU ASSIGN A VALUE TO A VARIABLE:

Example	Data Type	Try it
x = "Hello World"	str	Try it »
x = 20	int	Try it »
x = 20.5	float	Try it »
x = 1j	complex	Try it »
x = ["apple", "banana", "cherry"]	list	Try it »
x = ("apple", "banana", "cherry")	tuple	Try it »
x = range(6)	range	Try it »
x = {"name" : "John", "age" : 36}	dict	Try it »
x = {"apple", "banana", "cherry"}	set	Try it »
x = frozenset({"apple", "banana", "cherry"})	frozenset	Try it »
x = True	bool	Try it »
x = b"Hello"	bytes	Try it »
x = bytearray(5)	bytearray	Try it »
x = memoryview(bytes(5))	memoryview	Try it »
x = None	NoneType	Try it »

SETTING THE SPECIFIC DATA TYPE



IF YOU WANT TO SPECIFY THE DATA TYPE, YOU CAN USE THE FOLLOWING CONSTRUCTOR FUNCTIONS:

Example	Data Type	Try it
<code>x = str("Hello World")</code>	str	Try it »
<code>x = int(20)</code>	int	Try it »
<code>x = float(20.5)</code>	float	Try it »
<code>x = complex(1j)</code>	complex	Try it »
<code>x = list(("apple", "banana", "cherry"))</code>	list	Try it »
<code>x = tuple(("apple", "banana", "cherry"))</code>	tuple	Try it »
<code>x = range(6)</code>	range	Try it »
<code>x = dict(name="John", age=36)</code>	dict	Try it »
<code>x = set(("apple", "banana", "cherry"))</code>	set	Try it »
<code>x = frozenset(("apple", "banana", "cherry"))</code>	frozenset	Try it »
<code>x = bool(5)</code>	bool	Try it »
<code>x = bytes(5)</code>	bytes	Try it »
<code>x = bytearray(5)</code>	bytearray	Try it »
<code>x = memoryview(bytes(5))</code>	memoryview	Try it »

PYTHON NUMBERS



There are three numeric types in Python:

int

float

complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1      # int
y = 2.8    # float
z = 1j     # complex
```

To verify the type of any object in Python, use the `type()` function:

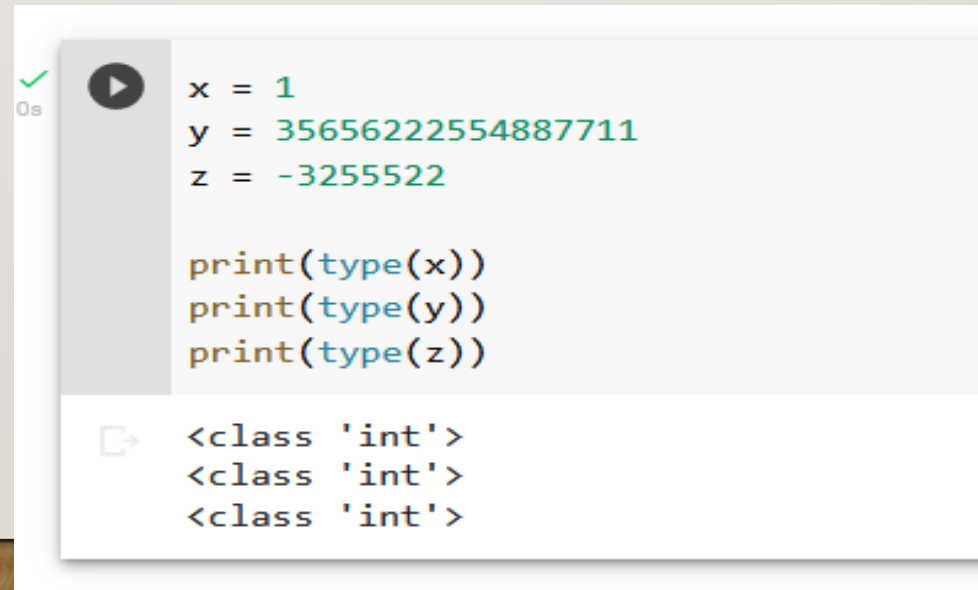
```
print(type(x))
print(type(y))
print(type(z))

<class 'int'>
<class 'float'>
<class 'complex'>
```

INT

- Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
- **Example**

Integers



```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

<class 'int'>
<class 'int'>
<class 'int'>

The image shows a Python code execution window. On the left, there is a green checkmark and a play button icon. The code defines three integer variables: x = 1, y = 35656222554887711, and z = -3255522. It then prints the type of each variable, which are all <class 'int'>.

FLOAT

- Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
- **Example**

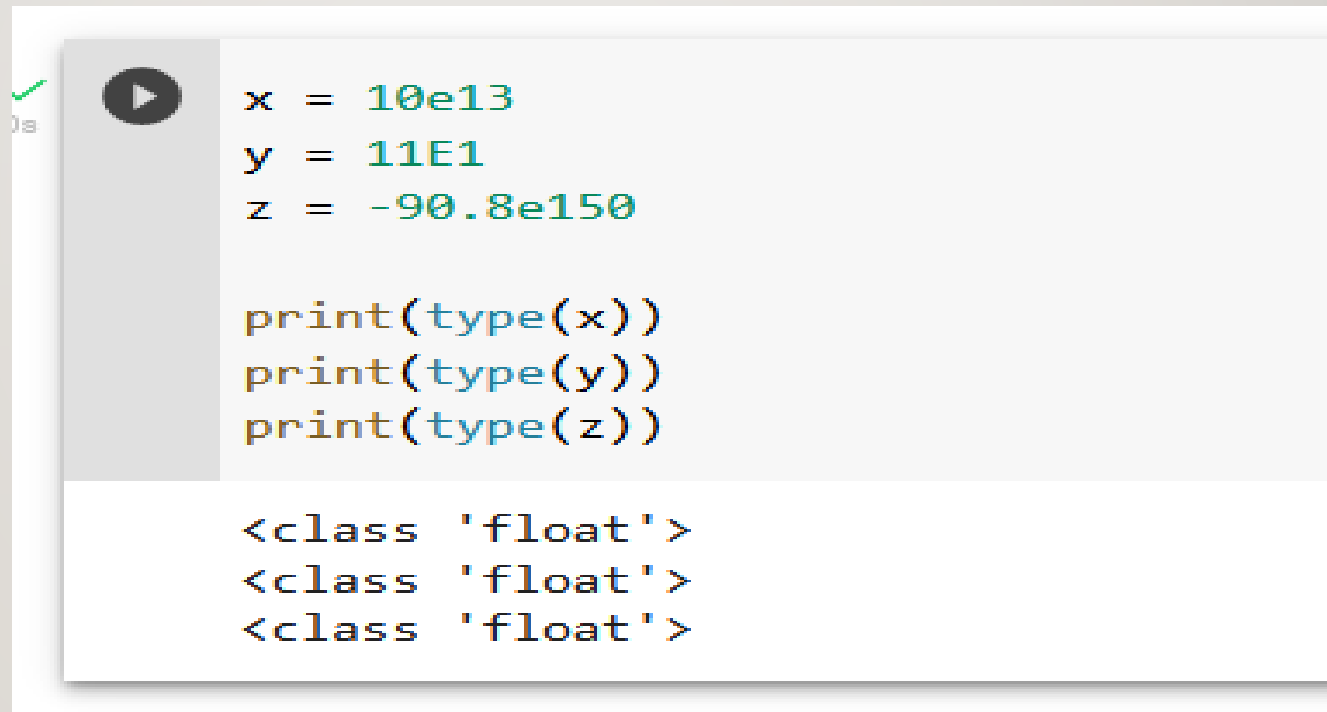
```
x = 1.13
y = 1.10
z = -25.13

print(type(x))
print(type(y))
print(type(z))

<class 'float'>
<class 'float'>
<class 'float'>
```

FLOAT CAN ALSO BE SCIENTIFIC NUMBERS WITH AN "E" TO INDICATE THE POWER OF 10.

- **Example**
- Floats:



```
x = 10e13
y = 11E1
z = -90.8e150

print(type(x))
print(type(y))
print(type(z))

<class 'float'>
<class 'float'>
<class 'float'>
```

The image shows a Python code execution window. On the left, there is a grey vertical bar with a play button icon. To the right of this bar, the code is displayed in a light blue font. The code defines three variables: x, y, and z, each assigned a scientific notation value. Below the assignments, there are three print statements to check the type of each variable. The output shows that all three variables are of type 'float'.

COMPLEX

- Complex numbers are written with a "j" as the imaginary part:
- **Example**
- Complex:

```
✓ 0s ▶ x = 1+8j  
y = 8j  
z = -8j  
  
print(type(x))  
print(type(y))  
print(type(z))  
  
↵ <class 'complex'>  
   <class 'complex'>  
   <class 'complex'>
```

TYPE CONVERSION

- You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:
- **Example**
- Convert from one type to another:

```
x = 13      # int
y = 1.10    # float
z = 8j      # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

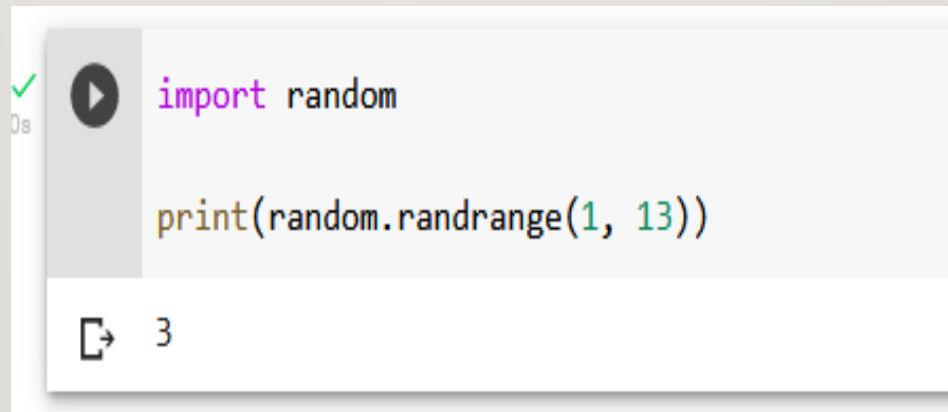
print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

```
13.0
1
(13+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

RANDOM NUMBER

- Python does not have a random() function to make a random number, but Python has a built-in module called random that can be used to make random numbers:
- **Example**
- Import the random module, and display a random number between 1 and 9:



```
import random

print(random.randrange(1, 13))
```

3

PYTHON CASTING



SPECIFY A VARIABLE TYPE



-
- There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, including its primitive types.
 - Casting in python is therefore done using constructor functions:
 - `int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)
 - `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
 - `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

EXAMPLE

- Integers:

```
✓  
0s ▶ x = int(1)    # x will be 1  
      y = int(1.8) # y will be 10  
      z = int("13") # z will be 8
```

Floats:

```
✓  
0s ▶ x = float(1)    # x will be 1.0  
      y = float(2.8)  # y will be 3.8  
      z = float("13")  # z will be 1.13  
      w = float("8.10") # w will be 8.20
```

Strings:

```
✓  
0s ▶ x = str("s1") # x will be 's1'  
      y = str(8)   # y will be '10'  
      z = str(13.0) # z will be '1.8'
```

PYTHON STRINGS



STRINGS

- Strings in python are surrounded by either single quotation marks, or double quotation marks.
- 'hello' is the same as "hello".

YOU CAN DISPLAY A STRING LITERAL WITH THE PRINT() FUNCTION:

- **Example**

A screenshot of a code editor showing two lines of Python code: `print("Hello")` and `print('Hello')`. The code is executed, and the output is displayed below it: `Hello` followed by `Hello` on a new line. The code is written in a monospaced font, with the function name `print` in blue and the string literals in red. The output is in a standard black font. The background of the code editor is light gray, and the output area is white.

```
print("Hello")
print('Hello')
```

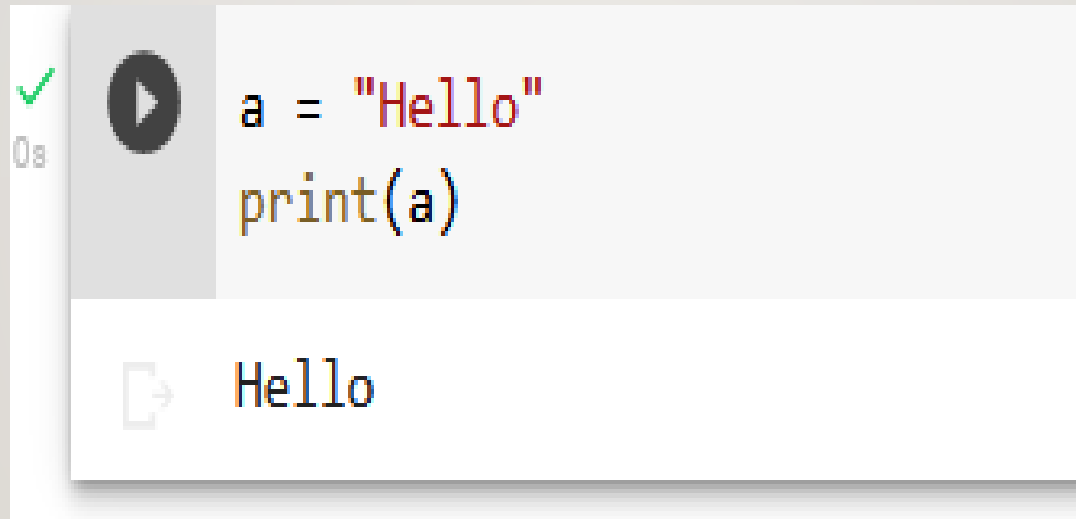
```
Hello
Hello
```

ASSIGN STRING TO A VARIABLE



ASSIGNING A STRING TO A VARIABLE IS DONE WITH THE VARIABLE NAME FOLLOWED BY AN EQUAL SIGN AND THE STRING:

- **Example**

A screenshot of a code editor or terminal window. On the left, there is a green checkmark and the text '0s'. Next to it is a play button icon. The main area contains two lines of Python code: 'a = "Hello"' and 'print(a)'. Below the code, there is a separate line showing the output 'Hello' preceded by a copy icon.

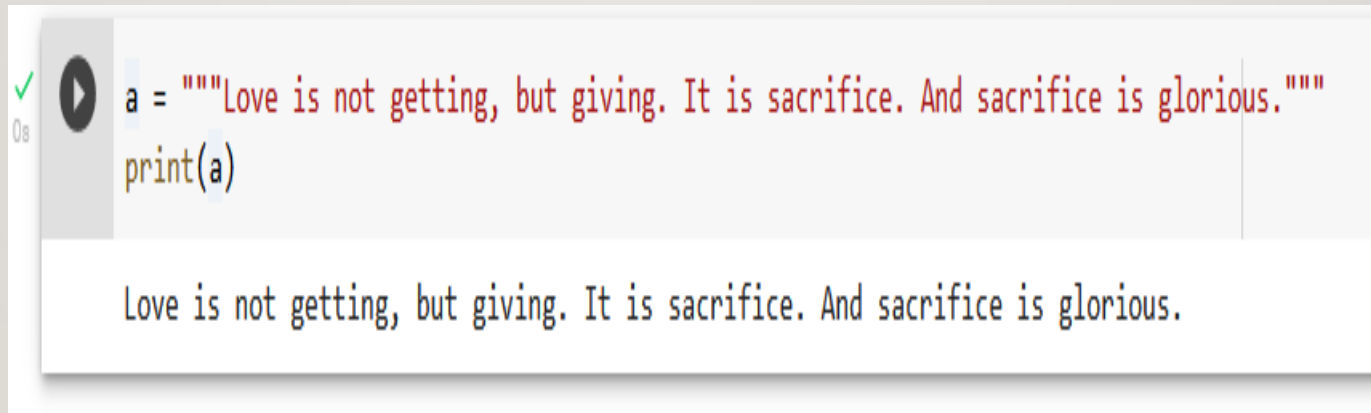
```
a = "Hello"
print(a)
```

→ Hello

MULTILINE STRINGS



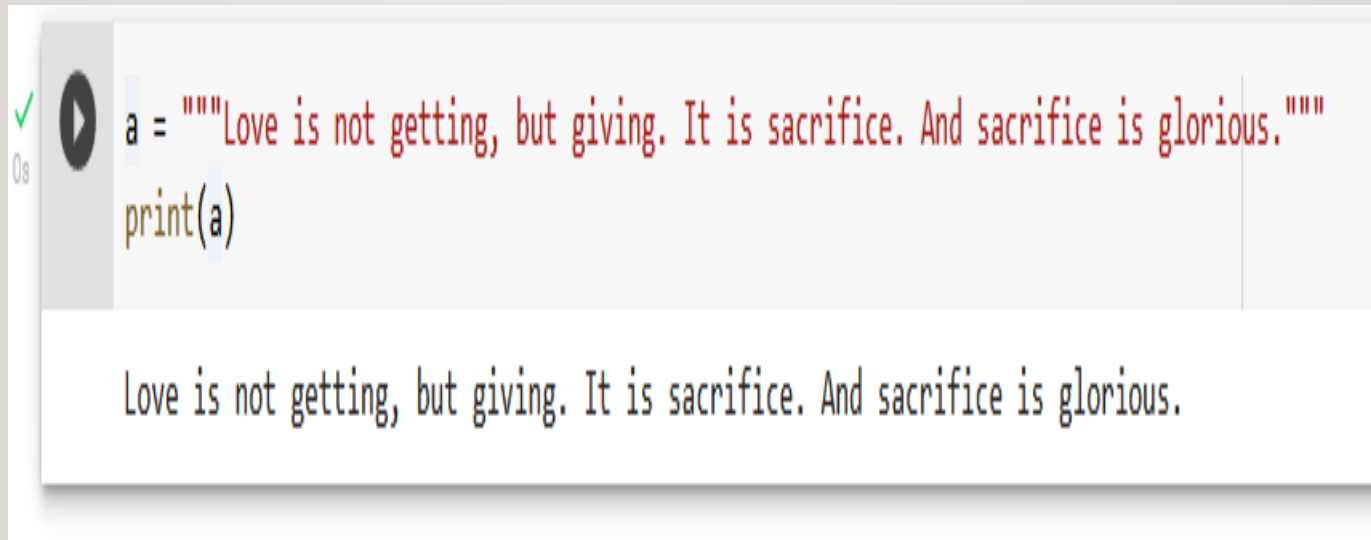
-
- You can assign a multiline string to a variable by using three quotes:
 - **Example**
 - You can use three double quotes:


A screenshot of a Python code execution environment. On the left, there is a green checkmark and a play button icon. The code is written in a light blue box: `a = """Love is not getting, but giving. It is sacrifice. And sacrifice is glorious."""` followed by `print(a)` on the next line. Below the code box, the output is displayed in a white box: `Love is not getting, but giving. It is sacrifice. And sacrifice is glorious.`

```
✓ 0s ▶ a = """Love is not getting, but giving. It is sacrifice. And sacrifice is glorious."""  
print(a)  
  
Love is not getting, but giving. It is sacrifice. And sacrifice is glorious.
```

OR THREE SINGLE QUOTES:

- **Example**

A screenshot of a Python code execution environment. On the left, there is a green checkmark and a play button icon. The code is written in a monospaced font: `a = """Love is not getting, but giving. It is sacrifice. And sacrifice is glorious."""` followed by `print(a)`. The output of the code is displayed below the code block: `Love is not getting, but giving. It is sacrifice. And sacrifice is glorious.`

```
✓  a = """Love is not getting, but giving. It is sacrifice. And sacrifice is glorious."""  
print(a)  
  
Love is not getting, but giving. It is sacrifice. And sacrifice is glorious.
```

STRINGS ARE ARRAYS



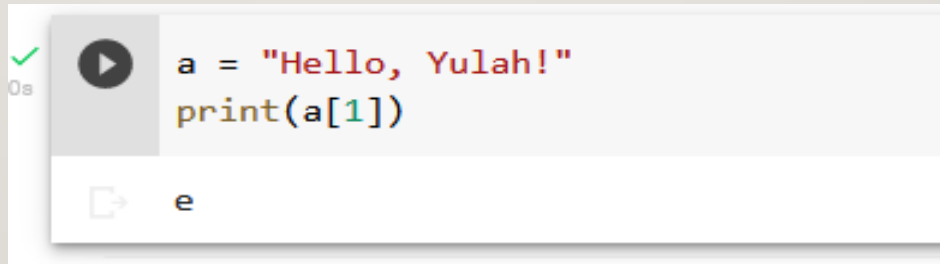
Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

A screenshot of a code execution environment. On the left, a green checkmark and '0s' indicate successful execution. A play button icon is next to the code. The code consists of two lines: `a = "Hello, Yulah!"` and `print(a[1])`. Below the code, the output 'e' is displayed next to a copy icon.

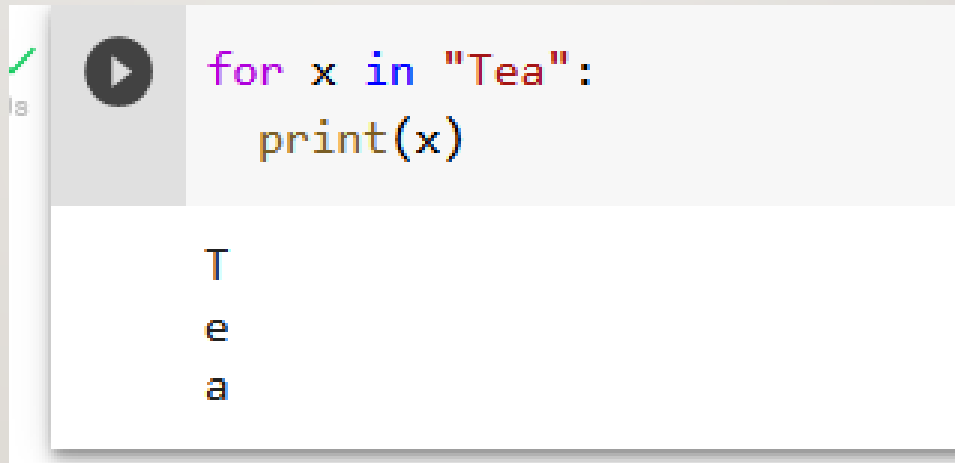
```
a = "Hello, Yulah!"  
print(a[1])  
  
e
```

LOOPING THROUGH A STRING



SINCE STRINGS ARE ARRAYS, WE CAN LOOP THROUGH THE CHARACTERS IN A STRING, WITH A FOR LOOP.

- **Example**
- Loop through the letters in the word “square”:



```
for x in "Tea":  
    print(x)
```

T
e
a

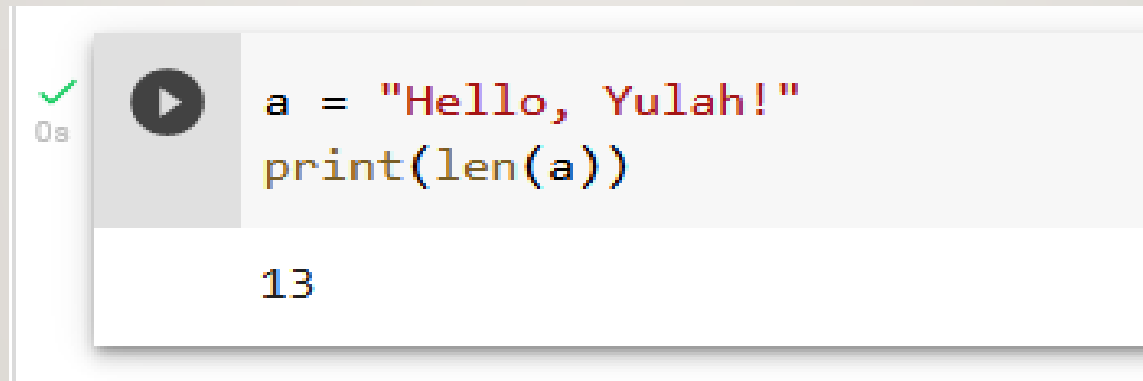
The image shows a code editor snippet with a play button icon on the left. The code is a Python for loop that iterates over the string "Tea" and prints each character. The output shows the characters 'T', 'e', and 'a' on separate lines.

STRING LENGTH



TO GET THE LENGTH OF A STRING, USE THE LEN() FUNCTION.

- **Example**
- The len() function returns the length of a string:



The image shows a code execution interface. On the left, there is a green checkmark and the text '0s'. To the right of this is a play button icon. The main area contains the following Python code: `a = "Hello, Yulah!"` and `print(len(a))`. Below the code, the output `13` is displayed.

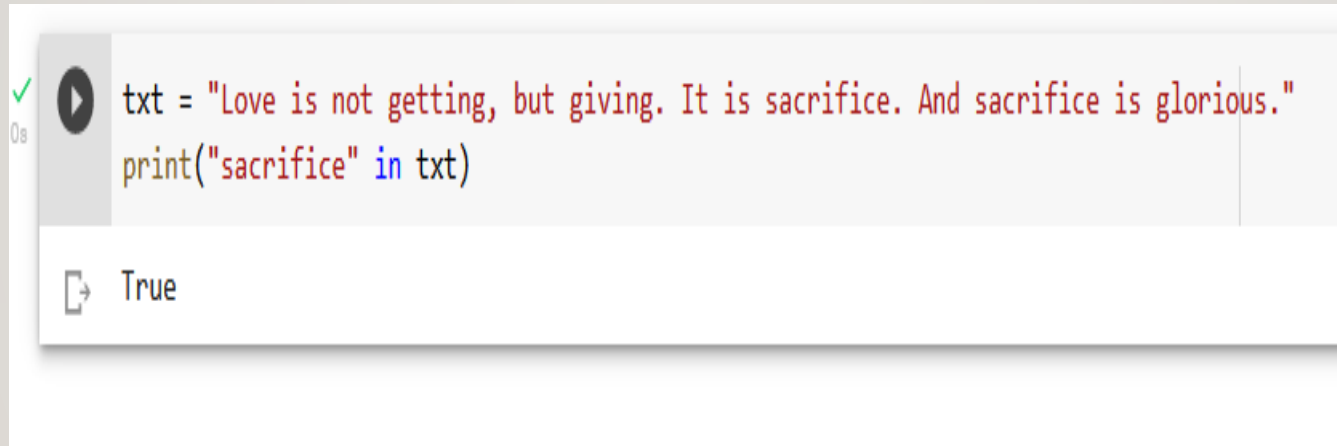
```
a = "Hello, Yulah!"  
print(len(a))  
  
13
```

CHECK STRING

- To check if a certain phrase or character is present in a string, we can use the keyword `in`.

EXAMPLE

- Check if “sacrifice” is present in the following text



```
✓ 0s ▶ txt = "Love is not getting, but giving. It is sacrifice. And sacrifice is glorious."  
print("sacrifice" in txt)  
→ True
```

The image shows a code execution environment. On the left, there is a green checkmark and the text '0s'. To the right of this is a play button icon. The main area contains two lines of Python code: `txt = "Love is not getting, but giving. It is sacrifice. And sacrifice is glorious."` and `print("sacrifice" in txt)`. Below the code, there is a result line showing a right-pointing arrow icon followed by the word `True`.

USE IT IN AN IF STATEMENT:

- **Example**
- Print only if “sacrifice” is present:

```
txt = "a goal without a plan is just a dream!"  
if "goal" in txt:  
    print("Yes, 'goal' is present.")
```

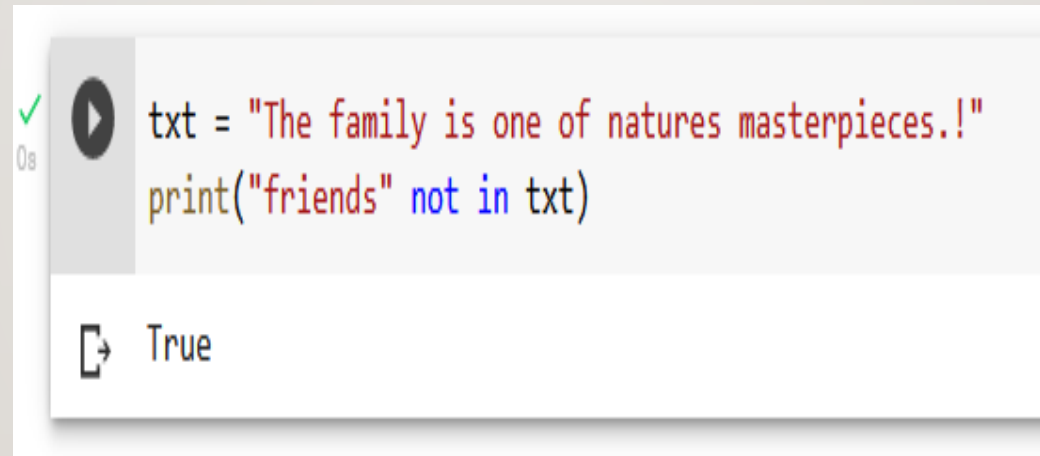
Yes, 'goal' is present.

CHECK IF NOT

- To check if a certain phrase or character is **NOT** present in a string, we can not in.

EXAMPLE

- Check if “**Family**” is NOT present in the following text:

A screenshot of a code execution environment. On the left, there is a green checkmark and a '0s' timer. Next to it is a play button icon. The code being executed is:

```
txt = "The family is one of natures masterpieces.!"  
print("friends" not in txt)
```

 Below the code, the output is shown as 'True' with a copy icon to its left.

```
txt = "The family is one of natures masterpieces.!"  
print("friends" not in txt)
```

True

USE IT IN AN **IF** STATEMENT:

- **Example**
- print only if “friendship” is NOT present:

```
txt = "The family is one of natures masterpieces!"  
if "friends" not in txt:  
    print("No, 'friends' is NOT present.")
```

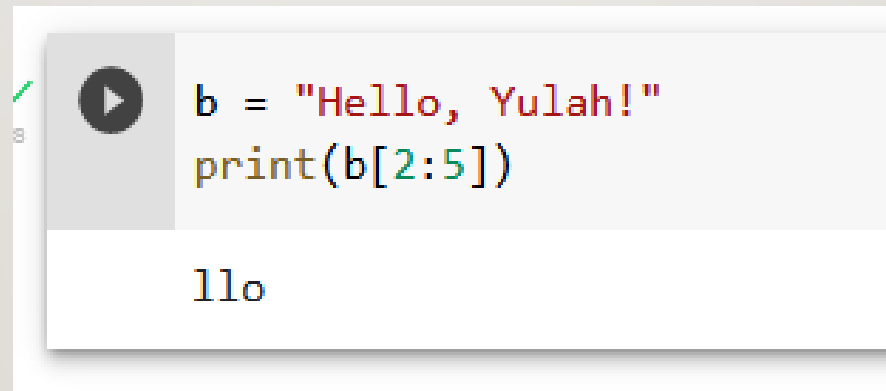
No, 'friends' is NOT present.

PYTHON - SLICING STRINGS



EXAMPLE

- Get the characters from position 2 to position 5 (not included):

A screenshot of a code editor or terminal window. It shows a Python code snippet with a play button icon on the left. The code defines a string 'b' as 'Hello, Yulah!' and prints the slice 'b[2:5]'. The output 'llo' is shown below the code. The background of the snippet is white with a light gray border and a subtle shadow.

```
b = "Hello, Yulah!"  
print(b[2:5])  
  
llo
```

SLICING

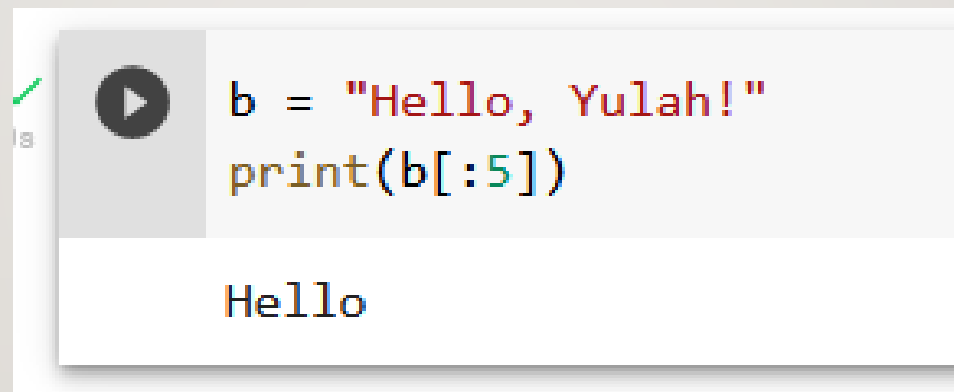
- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.

SLICE FROM THE START

- By leaving out the start index, the range will start at the first character:

EXAMPLE

- Get the characters from the start to position 5 (not included):

A screenshot of a code execution environment. On the left, there is a vertical sidebar with a green checkmark and a play button icon. The main area shows two lines of Python code: `b = "Hello, Yulah!"` and `print(b[:5])`. Below the code, the output `Hello` is displayed.

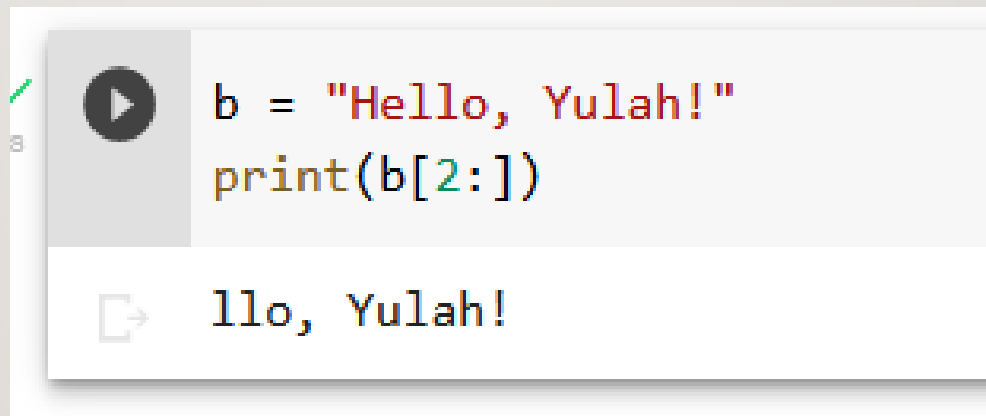
```
b = "Hello, Yulah!"  
print(b[:5])  
  
Hello
```

SLICE TO THE END

- By leaving out the *end* index, the range will go to the end:

EXAMPLE

- Get the characters from position 2, and all the way to the end:

A code execution snippet showing a Python string slicing operation. It includes a play button icon, the code lines, and the resulting output.

```
b = "Hello, Yulah!"  
print(b[2:])
```

llo, Yulah!

NEGATIVE INDEXING

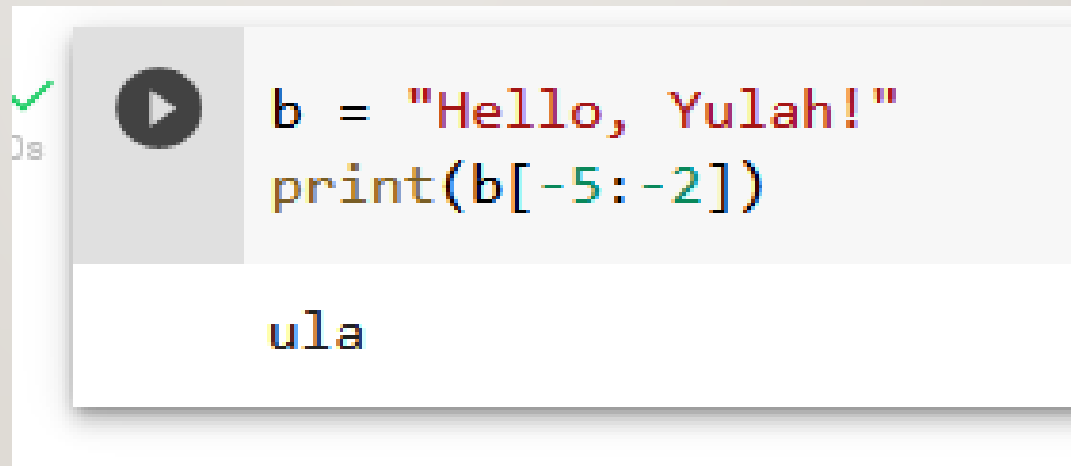
- Use negative indexes to start the slice from the end of the string:

EXAMPLE

- Get the characters:

From: "l" in "Yulah!" (position -5)

To, but not included: "h" in "Yulah!" (position -2)

A screenshot of a code execution environment. On the left, a green checkmark and the text '0s' are visible. The main area contains a play button icon in a grey box, followed by the Python code:

```
b = "Hello, Yulah!"  
print(b[-5:-2])
```

 Below the code, the output 'ula' is displayed.

```
b = "Hello, Yulah!"  
print(b[-5:-2])  
  
ula
```

PYTHON - MODIFY STRINGS

- Python has a set of built-in methods that you can use on strings.

UPPER CASE



EXAMPLE

- The upper() method returns the string in upper case:



```
a = "Hello, Yulah!"  
print(a.upper())
```



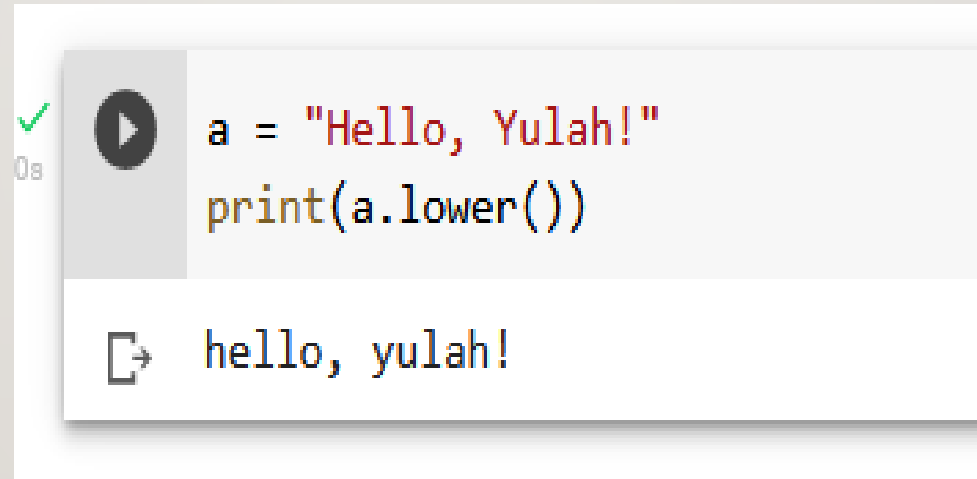
```
HELLO, YULAH!
```

LOWER CASE



EXAMPLE

- The lower() method returns the string in lower case:



A code execution snippet from a Python environment. It features a green checkmark and a '0s' timer on the left. The code block contains two lines: `a = "Hello, Yulah!"` and `print(a.lower())`. Below the code, the output `hello, yulah!` is displayed with a copy icon to its left.

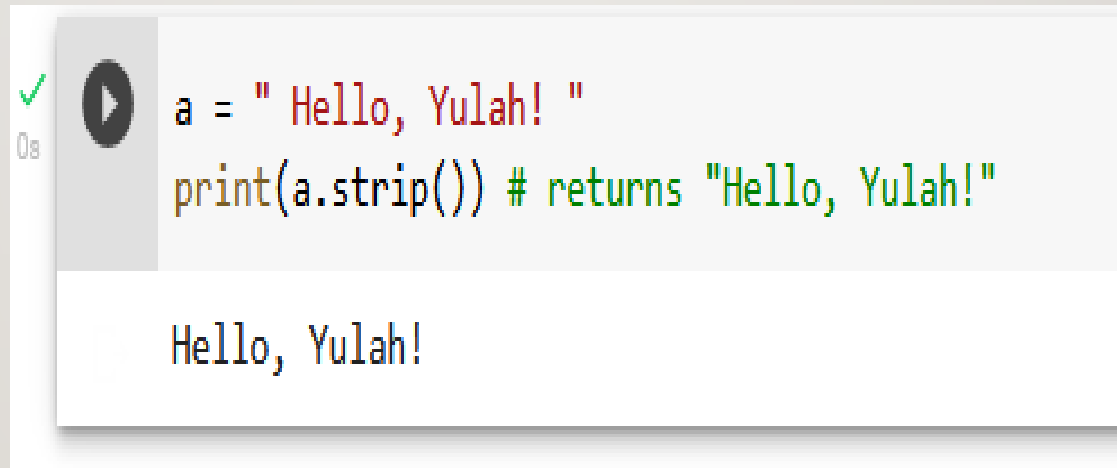
```
a = "Hello, Yulah!"  
print(a.lower())  
  
hello, yulah!
```

REMOVE WHITESPACE

- Whitespace is the space before and/or after the actual text, and very often you want to remove this space.

EXAMPLE

- The strip() method removes any whitespace from the beginning or the end:



A code execution snippet from a Python environment. It features a green checkmark and a play button icon on the left. The code defines a variable `a` with the value `" Hello, Yulah! "` and prints the result of `a.strip()`, which is `"Hello, Yulah!"`. The output of the code is displayed below the code block.

```
a = " Hello, Yulah! "  
print(a.strip()) # returns "Hello, Yulah!"
```

Hello, Yulah!

REPLACE STRING



EXAMPLE

- The `replace()` method replaces a string with another string:

A screenshot of a Python REPL (Read-Eval-Print Loop) window. On the left, there is a green checkmark and the text '0s'. In the center, there is a play button icon. To the right of the play button, the code 'a = "Hello, Yulah!"' and 'print(a.replace("H", "Y"))' is displayed in a monospaced font. Below the code, the output 'Yello, Yulah!' is shown.

```
a = "Hello, Yulah!"  
print(a.replace("H", "Y"))  
  
Yello, Yulah!
```

SPLIT STRING

- The `split()` method returns a list where the text between the specified separator becomes the list items.

EXAMPLE

- The `split()` method splits the string into substrings if it finds instances of the separator:

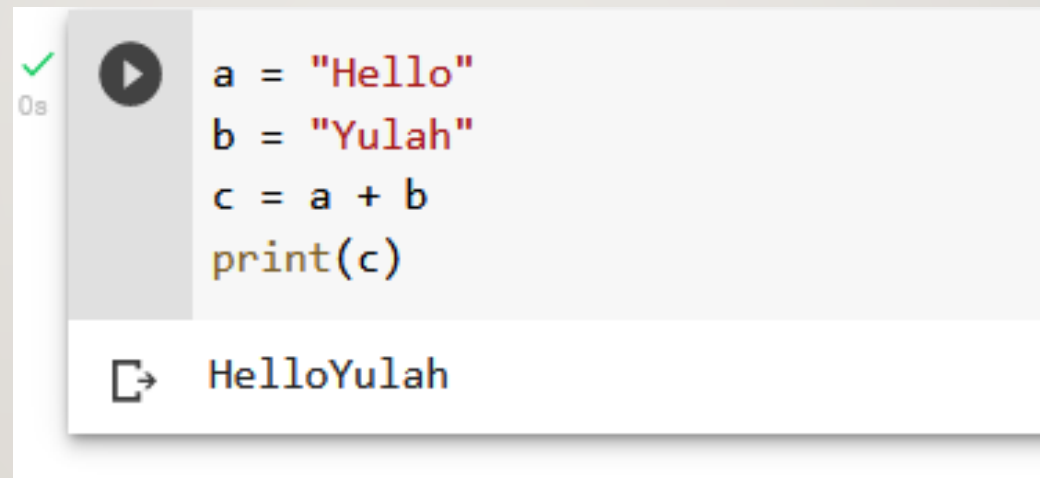
```
✓ 08 ▶ a = "Hello, Yulah!"  
    print(a.split(",")) # returns ['Hello', ' Yulah!']  
  
    ['Hello', ' Yulah!']
```

PYTHON - STRING CONCATENATION

- **String Concatenation**
- To concatenate, or combine, two strings you can use the + operator.

EXAMPLE

- Merge variable a with variable b into variable c:



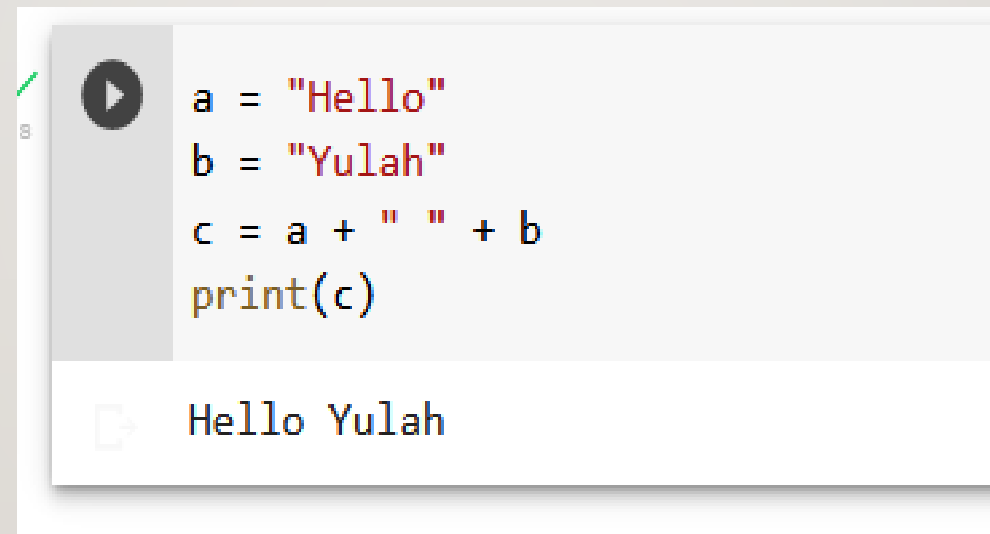
The image shows a code execution interface. On the left, there is a green checkmark and the text '0s'. In the center, there is a play button icon. To the right of the play button, the following code is displayed: `a = "Hello"`, `b = "Yulah"`, `c = a + b`, and `print(c)`. Below the code, there is a copy icon and the output 'HelloYulah'.

```
a = "Hello"
b = "Yulah"
c = a + b
print(c)
```

→ HelloYulah

EXAMPLE

- To add a space between them, add a " ":

A screenshot of a code editor or terminal window. On the left, there is a vertical sidebar with a green checkmark at the top and a play button icon. The main area contains Python code:

```
a = "Hello"
b = "Yulah"
c = a + " " + b
print(c)
```

 Below the code, the output "Hello Yulah" is displayed. The code is color-coded: strings are in red, variables in black, and the print function in blue. The output is in a monospaced font.

```
a = "Hello"
b = "Yulah"
c = a + " " + b
print(c)
```

Hello Yulah

PYTHON - FORMAT - STRINGS



STRING FORMAT



AS WE LEARNED IN THE PYTHON VARIABLES CHAPTER, WE CANNOT COMBINE STRINGS AND NUMBERS LIKE THIS:

- **Example**



```
age = 19
txt = "My name is Yulah, I am " + age
print(txt)
```

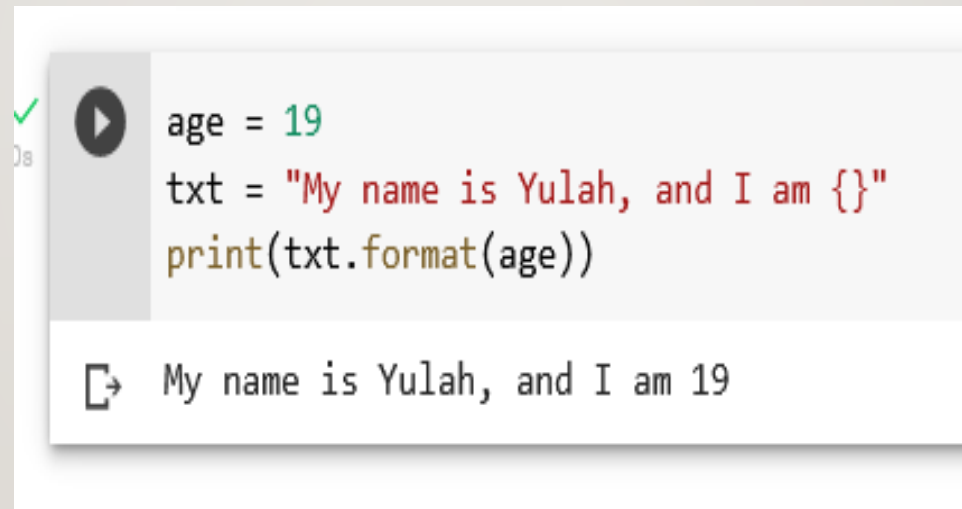
TypeError Traceback (most recent call last)
<ipython-input-18-1e876455dfef> in <module>
 1 age = 19
----> 2 txt = "My name is Yulah, I am " + age
 3 print(txt)

TypeError: can only concatenate str (not "int") to str

SEARCH STACK OVERFLOW

EXAMPLE

Use the `format()` method to insert numbers into strings:



```
age = 19
txt = "My name is Yulah, and I am {}"
print(txt.format(age))
```

My name is Yulah, and I am 19

The image shows a code execution environment. On the left, there is a green checkmark and a play button icon. The code is written in a monospaced font with syntax highlighting: 'age' is green, '19' is green, 'txt' is red, 'My name is Yulah, and I am {}' is red, 'print' is brown, and 'txt.format' is brown. Below the code, the output is displayed in a monospaced font: 'My name is Yulah, and I am 19'.

THE FORMAT() METHOD TAKES UNLIMITED NUMBER OF ARGUMENTS, AND ARE PLACED INTO THE RESPECTIVE PLACEHOLDERS:

- **Example**

```
✓ ▶ quantity = 8
    itemno = 550
    price = 50.80
    myorder = "I want {} pieces of item {} for {} dollars."
    print(myorder.format(quantity, itemno, price))
```

📄 I want 8 pieces of item 550 for 50.8 dollars.

You can use index numbers {0} to be sure the arguments are placed in the correct placeholders.

- **Example**

```
✓ 0s ▶ quantity = 8
    itemno = 550
    price = 50.80
    myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
    print(myorder.format(quantity, itemno, price))
```

📄 I want to pay 50.8 dollars for 8 pieces of item 550.

PYTHON - ESCAPE CHARACTERS



ESCAPE CHARACTER

To insert characters that are illegal in a string, use an escape character.
An escape character is a backslash \ followed by the character you want to insert.

AN EXAMPLE OF AN ILLEGAL CHARACTER IS A DOUBLE QUOTE INSIDE A STRING THAT IS SURROUNDED BY DOUBLE QUOTES:

- **Example**

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

A screenshot of a Jupyter Notebook interface showing a syntax error. The top part shows a code cell with the text `txt = "We are the so-called "Vikings" from the north."`. The string is underlined with a red wavy line, indicating an error. Below the code cell, the error message is displayed: `File "<ipython-input-22-56cdf4283a8e>", line 1` followed by the code `txt = "We are the so-called "Vikings" from the north."` with a red caret under the closing double quote. Below the code, the error message `SyntaxError: invalid syntax` is shown. At the bottom of the error message box, there is a button that says `SEARCH STACK OVERFLOW`.

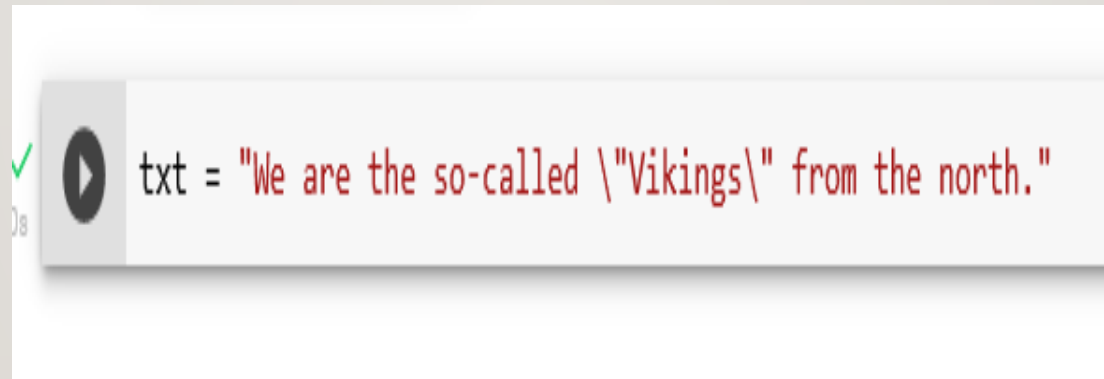
```
txt = "We are the so-called "Vikings" from the north."

File "<ipython-input-22-56cdf4283a8e>", line 1
    txt = "We are the so-called "Vikings" from the north."
                                   ^
SyntaxError: invalid syntax

SEARCH STACK OVERFLOW
```

EXAMPLE

- The escape character allows you to use double quotes when you normally would not be allowed:



```
txt = "We are the so-called \"Vikings\" from the north."
```

A code editor snippet showing a string with escaped double quotes. The text is displayed in a monospaced font, with the string content in red and the code structure in black. A play button icon is visible on the left side of the code block.

ESCAPE CHARACTERS

- Other escape characters used in Python:

Code	Result	Try it
\'	Single Quote	Try it »
\\	Backslash	Try it »
\n	New Line	Try it »
\r	Carriage Return	Try it »
\t	Tab	Try it »
\b	Backspace	Try it »
\f	Form Feed	
\ooo	Octal value	Try it »
\xhh	Hex value	Try it »

PYTHON - STRING METHODS



STRING METHODS

- Python has a set of built-in methods that you can use on strings.

Method	Description
<code>capitalize()</code>	Converts the first character to upper case
<code>casefold()</code>	Converts string into lower case
<code>center()</code>	Returns a centered string
<code>count()</code>	Returns the number of times a specified value occurs in a string
<code>encode()</code>	Returns an encoded version of the string
<code>endswith()</code>	Returns true if the string ends with the specified value
<code>expandtabs()</code>	Sets the tab size of the string
<code>find()</code>	Searches the string for a specified value and returns the position of where it was found
<code>format()</code>	Formats specified values in a string
<code>format_map()</code>	Formats specified values in a string
<code>index()</code>	Searches the string for a specified value and returns the position of where it was found
<code>isalnum()</code>	Returns True if all characters in the string are alphanumeric
<code>isalpha()</code>	Returns True if all characters in the string are in the alphabet
<code>isdecimal()</code>	Returns True if all characters in the string are decimals
<code>isdigit()</code>	Returns True if all characters in the string are digits
<code>isidentifier()</code>	Returns True if the string is an identifier
<code>islower()</code>	Returns True if all characters in the string are lower case
<code>isnumeric()</code>	Returns True if all characters in the string are numeric
<code>isprintable()</code>	Returns True if all characters in the string are printable
<code>isspace()</code>	Returns True if all characters in the string are whitespaces
<code>istitle()</code>	Returns True if the string follows the rules of a title
<code>isupper()</code>	Returns True if all characters in the string are upper case
<code>join()</code>	Joins the elements of an iterable to the end of the string

<code>ljust()</code>	Returns a left justified version of the string
<code>lower()</code>	Converts a string into lower case
<code>lstrip()</code>	Returns a left trim version of the string
<code>maketrans()</code>	Returns a translation table to be used in translations
<code>partition()</code>	Returns a tuple where the string is parted into three parts
<code>replace()</code>	Returns a string where a specified value is replaced with a specified value
<code>rfind()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rindex()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rjust()</code>	Returns a right justified version of the string
<code>rpartition()</code>	Returns a tuple where the string is parted into three parts
<code>rsplit()</code>	Splits the string at the specified separator, and returns a list
<code>rstrip()</code>	Returns a right trim version of the string
<code>split()</code>	Splits the string at the specified separator, and returns a list
<code>splittines()</code>	Splits the string at line breaks and returns a list
<code>startswith()</code>	Returns true if the string starts with the specified value
<code>strip()</code>	Returns a trimmed version of the string
<code>swapcase()</code>	Swaps cases, lower case becomes upper case and vice versa
<code>title()</code>	Converts the first character of each word to upper case
<code>translate()</code>	Returns a translated string
<code>upper()</code>	Converts a string into upper case
<code>zfill()</code>	Fills the string with a specified number of 0 values at the beginning

Method	Description
<u>capitalize()</u>	Converts the first character to upper case
<u>casefold()</u>	Converts string into lower case
<u>center()</u>	Returns a centered string
<u>count()</u>	Returns the number of times a specified value occurs in a string
<u>encode()</u>	Returns an encoded version of the string
<u>endswith()</u>	Returns true if the string ends with the specified value
<u>expandtabs()</u>	Sets the tab size of the string
<u>find()</u>	Searches the string for a specified value and returns the position of where it was found
<u>format()</u>	Formats specified values in a string
<u>format_map()</u>	Formats specified values in a string
<u>index()</u>	Searches the string for a specified value and returns the position of where it was found
<u>isalnum()</u>	Returns True if all characters in the string are alphanumeric
<u>isalpha()</u>	Returns True if all characters in the string are in the alphabet
<u>isdecimal()</u>	Returns True if all characters in the string are decimals
<u>isdigit()</u>	Returns True if all characters in the string are digits
<u>isidentifier()</u>	Returns True if the string is an identifier
<u>islower()</u>	Returns True if all characters in the string are lower case
<u>isnumeric()</u>	Returns True if all characters in the string are numeric
<u>isprintable()</u>	Returns True if all characters in the string are printable
<u>isspace()</u>	Returns True if all characters in the string are whitespaces
<u>istitle()</u>	Returns True if the string follows the rules of a title
<u>isupper()</u>	Returns True if all characters in the string are upper case
<u>join()</u>	Joins the elements of an iterable to the end of the string

<code>ljust()</code>	Returns a left justified version of the string
<code>lower()</code>	Converts a string into lower case
<code>lstrip()</code>	Returns a left trim version of the string
<code>maketrans()</code>	Returns a translation table to be used in translations
<code>partition()</code>	Returns a tuple where the string is parted into three parts
<code>replace()</code>	Returns a string where a specified value is replaced with a specified value
<code>rfind()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rindex()</code>	Searches the string for a specified value and returns the last position of where it was found
<code>rjust()</code>	Returns a right justified version of the string
<code>rpartition()</code>	Returns a tuple where the string is parted into three parts
<code>rsplit()</code>	Splits the string at the specified separator, and returns a list
<code>rstrip()</code>	Returns a right trim version of the string
<code>split()</code>	Splits the string at the specified separator, and returns a list
<code>splitlines()</code>	Splits the string at line breaks and returns a list
<code>startswith()</code>	Returns true if the string starts with the specified value
<code>strip()</code>	Returns a trimmed version of the string
<code>swapcase()</code>	Swaps cases, lower case becomes upper case and vice versa
<code>title()</code>	Converts the first character of each word to upper case
<code>translate()</code>	Returns a translated string
<code>upper()</code>	Converts a string into upper case
<code>zfill()</code>	Fills the string with a specified number of 0 values at the beginning

PYTHON BOOLEANS



BOOLEAN VALUES

- In programming you often need to know if an expression is True or False.
- You can evaluate any expression in Python, and get one of two answers, True or False

WHEN YOU COMPARE TWO VALUES, THE EXPRESSION IS EVALUATED AND PYTHON RETURNS THE BOOLEAN ANSWER:

- **Example**

✓
0s

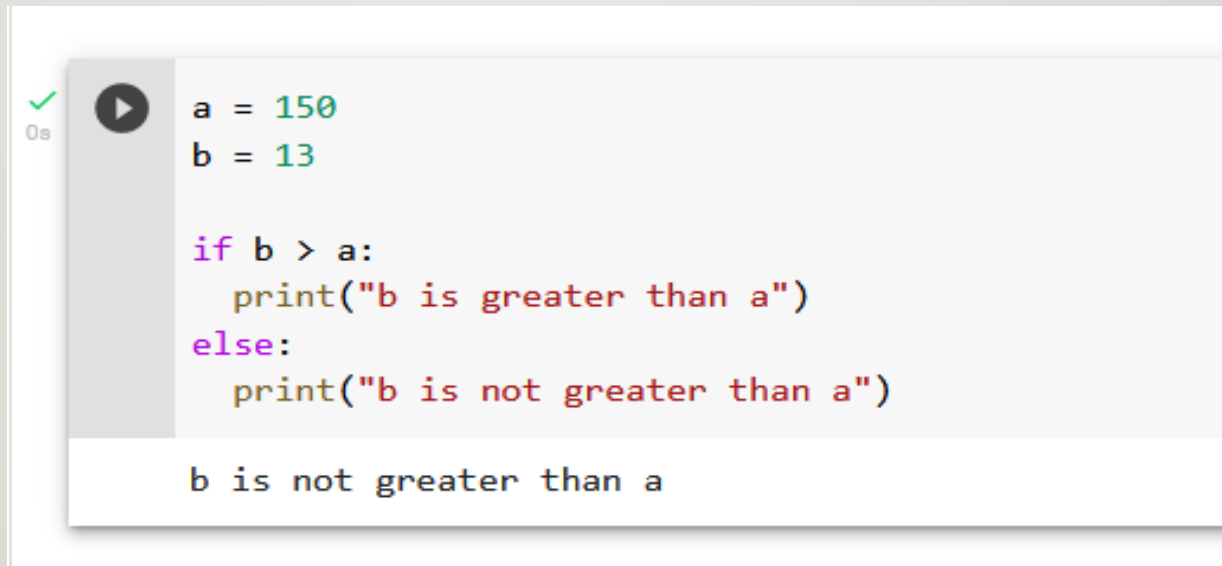


```
print(13 > 1)  
print(13 == 1)  
print(13 < 1)
```

```
True  
False  
False
```


WHEN YOU RUN A CONDITION IN AN IF STATEMENT, PYTHON RETURNS TRUE OR FALSE:

- **Example**
- Print a message based on whether the condition is True or False:

A screenshot of a Python code editor showing a script execution. On the left, there is a green checkmark and a play button icon. The code is as follows:

```
a = 150
b = 13

if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

Below the code, the output of the script is displayed: "b is not greater than a".

```
b is not greater than a
```

EVALUATE VALUES AND VARIABLES

- The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return,

EXAMPLE

- Evaluate a string and a number:

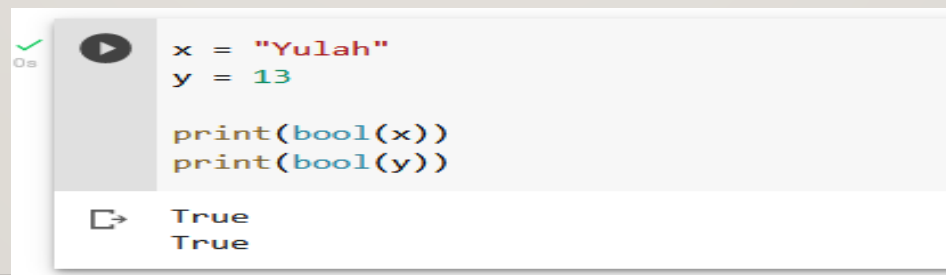


```
✓ 0s ▶ print(bool("Yulah"))  
      print(bool(13))  
  
📄 True  
   True
```

A code execution snippet showing two lines of Python code: `print(bool("Yulah"))` and `print(bool(13))`. The code is executed successfully, as indicated by a green checkmark and '0s' in the top left. The output, shown in a separate box below the code, is two lines of 'True'.

Example

Evaluate two variables:



```
✓ 0s ▶ x = "Yulah"  
      y = 13  
  
      print(bool(x))  
      print(bool(y))  
  
📄 True  
   True
```

A code execution snippet showing variable assignment and boolean conversion. The code is: `x = "Yulah"`, `y = 13`, `print(bool(x))`, and `print(bool(y))`. The code is executed successfully, as indicated by a green checkmark and '0s' in the top left. The output, shown in a separate box below the code, is two lines of 'True'.

MOST VALUES ARE TRUE



Almost any value is evaluated to True if it has some sort of content.

Any string is True, except empty strings.

Any number is True, except 0.

Any list, tuple, set, and dictionary are True, except empty ones.

Example

The following will return True:

A screenshot of a Python REPL (Read-Eval-Print Loop) window. On the left, there is a green checkmark and a '0s' timer. A play button icon is next to the input code. The code consists of three lines: 'bool("abc")', 'bool(123)', and 'bool(["Tea", "Water", "Coffee"])'. The output, 'True', is displayed below the code, preceded by a copy icon.

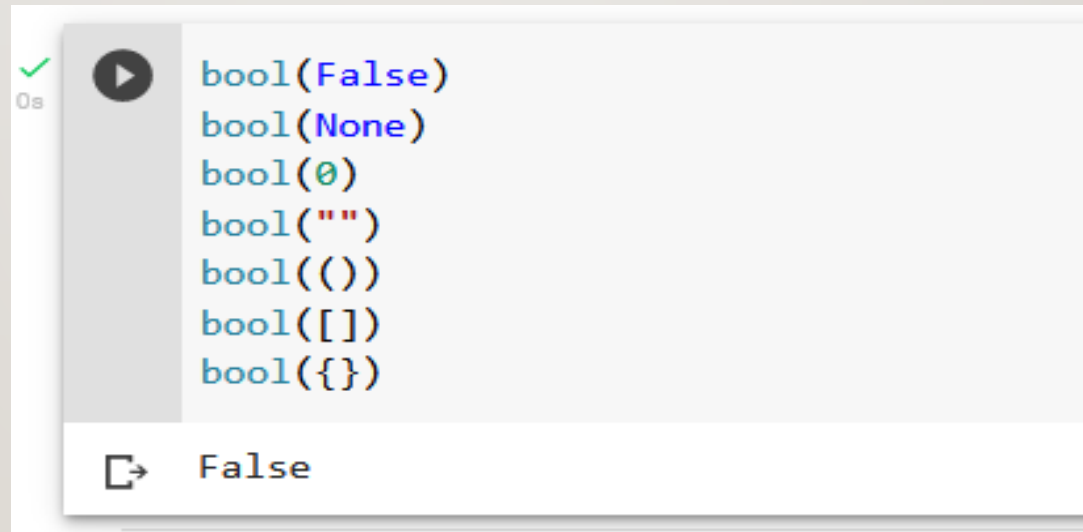
```
bool("abc")  
bool(123)  
bool(["Tea", "Water", "Coffee"])  
  
True
```


SOME VALUES ARE FALSE



IN FACT, THERE ARE NOT MANY VALUES THAT EVALUATE TO FALSE, EXCEPT EMPTY VALUES, SUCH AS (), [], {}, "", THE NUMBER 0, AND THE VALUE NONE. AND OF COURSE THE VALUE FALSE EVALUATES TO FALSE.

- **Example**
- The following will return False:



A screenshot of a Python REPL (Read-Eval-Print Loop) window. On the left, there is a green checkmark and the text '0s'. A play button icon is next to the code input area. The code input area contains the following lines of Python code: `bool(False)`, `bool(None)`, `bool(0)`, `bool("")`, `bool(())`, `bool([])`, and `bool({})`. Below the code input area, there is a button with a right-pointing arrow icon, and next to it, the output 'False' is displayed.

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

False

ONE MORE VALUE, OR OBJECT IN THIS CASE, EVALUATES TO FALSE, AND THAT IS IF YOU HAVE AN OBJECT THAT IS MADE FROM A CLASS WITH A `__len__` FUNCTION THAT RETURNS 0 OR FALSE:

- **Example**

```
class myclass():  
    def __len__(self):  
        return 0  
  
myobj = myclass()  
print(bool(myobj))
```

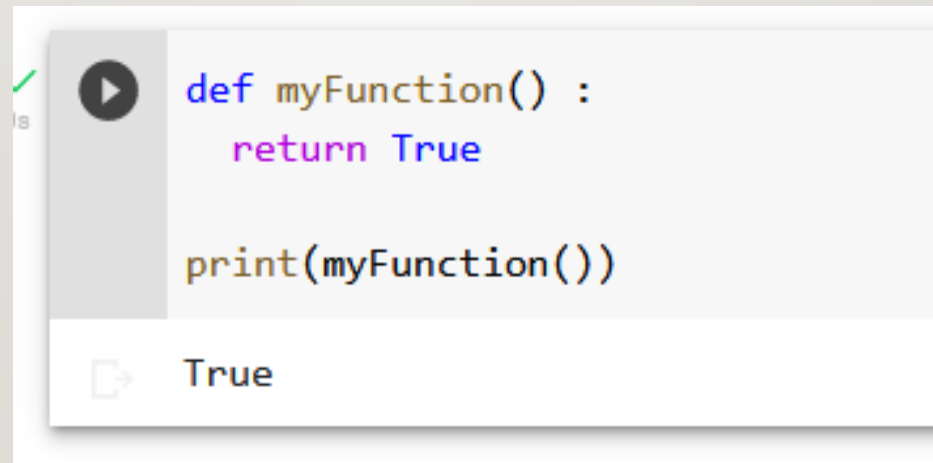
False

FUNCTIONS CAN RETURN A BOOLEAN



YOU CAN CREATE FUNCTIONS THAT RETURNS A BOOLEAN VALUE:

- **Example**
- Print the answer of a function:



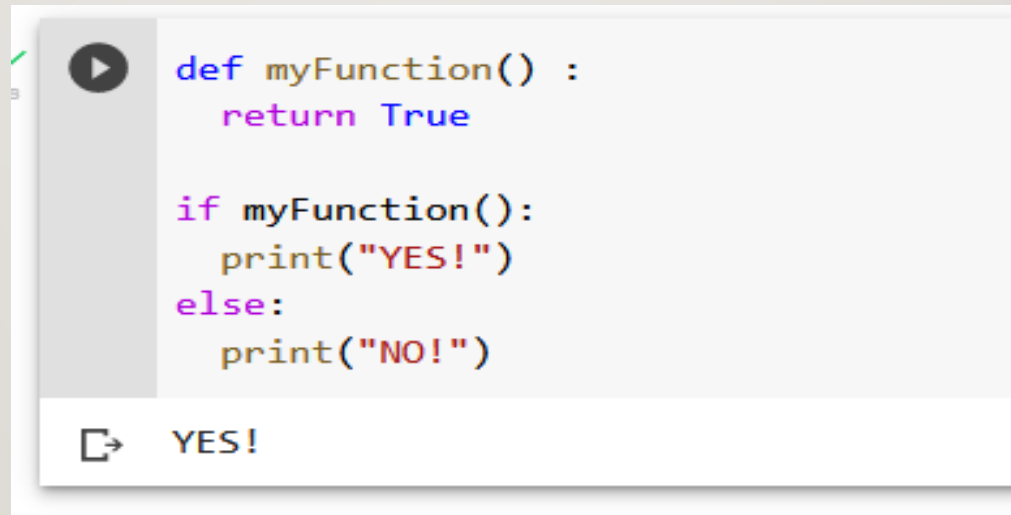
```
def myFunction() :  
    return True  
  
print(myFunction())
```

True

The image shows a snippet of Python code in a light gray editor. The code defines a function named `myFunction()` that returns the boolean value `True`. Below the function definition, the function is called using `print(myFunction())`. The output of the code, `True`, is displayed in a white box at the bottom of the snippet. A play button icon is visible on the left side of the code editor.

YOU CAN EXECUTE CODE BASED ON THE BOOLEAN ANSWER OF A FUNCTION:

- **Example**
- Print "YES!" if the function returns True, otherwise print "NO!":

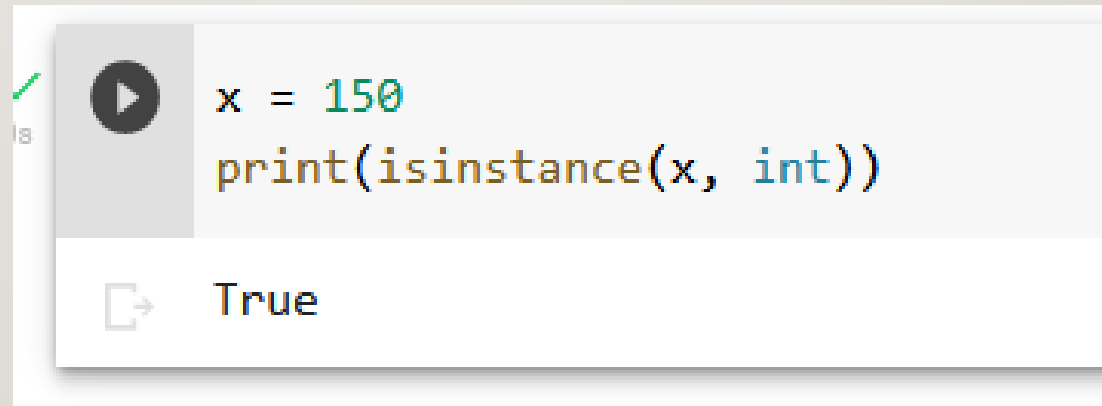


```
def myFunction() :  
    return True  
  
if myFunction():  
    print("YES!")  
else:  
    print("NO!")
```

YES!

PYTHON ALSO HAS MANY BUILT-IN FUNCTIONS THAT RETURN A BOOLEAN VALUE, LIKE THE `ISINSTANCE()` FUNCTION, WHICH CAN BE USED TO DETERMINE IF AN OBJECT IS OF A CERTAIN DATA TYPE:

- **Example**
- Check if an object is an integer or not:



```
x = 150
print(isinstance(x, int))
```

True

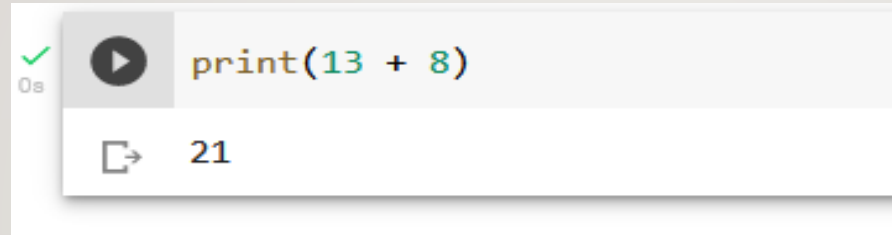
PYTHON OPERATORS





OPERATORS ARE USED TO PERFORM OPERATIONS ON VARIABLES AND VALUES.

IN THE EXAMPLE BELOW,WE USE THE + OPERATOR TO ADD TOGETHER TWO VALUES:

- **Example**

A screenshot of a code editor showing a Python command being executed. The command is `print(13 + 8)`. To the left of the code is a play button icon. Below the code, the output `21` is displayed next to a copy icon. A green checkmark and the text '0s' are visible in the top left corner of the execution area.

```
✓ 0s  print(13 + 8)  
 21
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

PYTHON ARITHMETIC OPERATORS

- Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$

PYTHON ASSIGNMENT OPERATORS

- Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

PYTHON COMPARISON OPERATORS

- Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

PYTHON LOGICAL OPERATORS

- Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	<code>x < 5 and x < 10</code>
or	Returns True if one of the statements is true	<code>x < 5 or x < 4</code>
not	Reverse the result, returns False if the result is true	<code>not(x < 5 and x < 10)</code>

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	<code>x is y</code>
is not	Returns True if both variables are not the same object	<code>x is not y</code>

PYTHON MEMBERSHIP OPERATORS

- Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

PYTHON LISTS



List

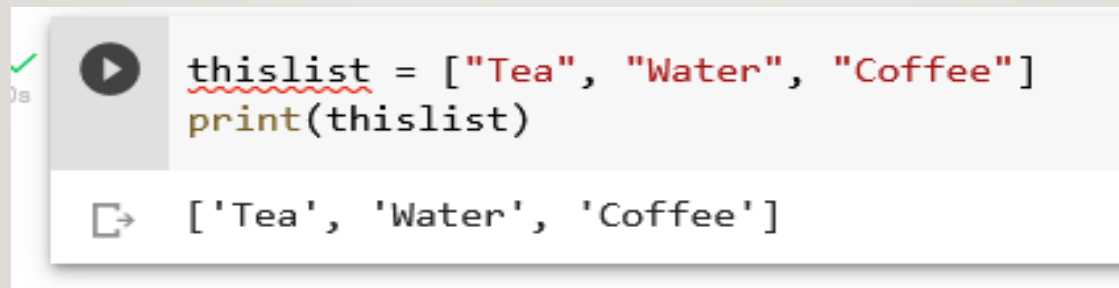
Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage.

Lists are created using square brackets:

EXAMPLE

- Create a List:



```
08 thislist = ["Tea", "Water", "Coffee"]  
    print(thislist)
```

['Tea', 'Water', 'Coffee']

The image shows a code execution snippet. On the left, there is a green checkmark and the number '08'. The code consists of two lines: `thislist = ["Tea", "Water", "Coffee"]` and `print(thislist)`. The variable `thislist` is underlined with a red wavy line. Below the code, the output is displayed as `['Tea', 'Water', 'Coffee']`.

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

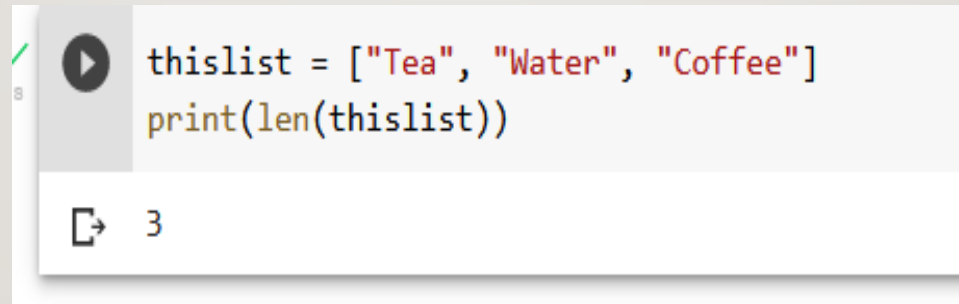
Lists allow duplicate values:

```
✓ 8 ▶ thislist = ["Tea", "Water", "Coffee", "Juice", "Milk"]  
    print(thislist)  
    ▶ ['Tea', 'Water', 'Coffee', 'Juice', 'Milk']
```

List Length

To determine how many items a list has, use the `len()` function:

- **Example**
- Print the number of items in the list:



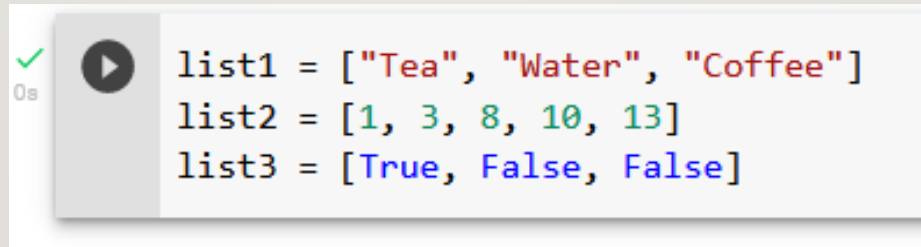
```
✓ 8 ▶ thislist = ["Tea", "Water", "Coffee"]  
    print(len(thislist))  
    ↵ 3
```

The image shows a code editor snippet with a green checkmark and line number 8. The code defines a list `thislist` with three elements: "Tea", "Water", and "Coffee". It then prints the length of the list using `len(thislist)`. The output, shown below the code, is the number 3.

LIST ITEMS - DATA TYPES

LIST ITEMS CAN BE OF ANY DATA TYPE:

- **Example**
- String, int and boolean data types:

A code execution snippet showing three lines of Python code. To the left of the code is a green checkmark, a play button icon, and the text '0s'.

```
list1 = ["Tea", "Water", "Coffee"]  
list2 = [1, 3, 8, 10, 13]  
list3 = [True, False, False]
```

A LIST CAN CONTAIN DIFFERENT DATA TYPES:

- **Example**
- A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "female"]
```

type()

From Python's perspective, lists are defined as objects with the data type 'list':

```
<class 'list'>
```


EXAMPLE

WHAT IS THE DATA TYPE OF A LIST?

```
mylist = ["Tea", "Water", "Coffee"]
print(type(mylist))
```

<class 'list'>

The list() Constructor

It is also possible to use the list() constructor when creating a new list.

Example

Using the list() constructor to make a List:

```
thislist = list(("Tea", "Water", "Coffee")) # note the double round-brackets
print(thislist)
```

['Tea', 'Water', 'Coffee']

PYTHON - ACCESS LIST ITEMS

- **Access Items**
- List items are indexed and you can access them by referring to the index number:
- **Example**
- Print the second item of the list:



```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      print(thislist[1])  
      ↗ Water
```

The image shows a code execution snippet. On the left, there is a green checkmark and the text '0s'. To the right of this is a play button icon. The code being executed is `thislist = ["Tea", "Water", "Coffee"]` followed by `print(thislist[1])`. Below the code, the output is shown as 'Water' with a small icon of a document and an arrow pointing to it.

Negative Indexing

Negative indexing means start from the end

-1 refers to the last item, -2 refers to the second last item etc.

- **Example**

- Print the last item of the list:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      print(thislist[-1])  
Coffee
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

Example

Return the third, fourth, and fifth item:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Juice", "Milk", "Wine", "Coke"]  
      print(thislist[2:5])  
['Coffee', 'Juice', 'Milk']
```

BY LEAVING OUT THE START VALUE, THE RANGE WILL START AT THE FIRST ITEM:

- **Example**
- This example returns the items from the beginning to, but NOT including, "kiwi":

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Juice", "Milk", "Wine", "Coke"]  
      print(thislist[:4])  
  
📄 ['Tea', 'Water', 'Coffee', 'Juice']
```

By leaving out the end value, the range will go on to the end of the list:

Example

This example returns the items from "cherry" to the end:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Juice", "Milk", "Wine", "Coke"]  
      print(thislist[2:])  
  
📄 ['Coffee', 'Juice', 'Milk', 'Wine', 'Coke']
```

RANGE OF NEGATIVE INDEXES

SPECIFY NEGATIVE INDEXES IF YOU WANT TO START THE SEARCH FROM THE END OF THE LIST:

- **Example**
- This example returns the items from "orange" (-4) to, but NOT including "mango" (-1):

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Juice", "Milk", "Wine", "Coke"]  
      print(thislist[-4:-1])  
→ ['Juice', 'Milk', 'Wine']
```

Check if Item Exists

To determine if a specified item is present in a list use the in keyword:

Example

Check if "apple" is present in the list:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      if "tea" in thislist:  
          print("Yes, 'Tea' is in the drinks list")
```


PYTHON - CHANGE LIST ITEMS

- **Change Item Value**
- To change the value of a specific item, refer to the index number:
- **Example**
- Change the second item:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      thislist[1] = "blackcurrant"  
      print(thislist)  
📄 ['Tea', 'blackcurrant', 'Coffee']
```

Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

Example

Change the values “tea” and “coffee” with the values “blackcurrant” and “wine”:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Juice", "Milk", "Wine", "Coke"]  
      thislist[1:3] = ["blackcurrant", "wine"]  
      print(thislist)  
📄 ['Tea', 'blackcurrant', 'wine', 'Juice', 'Milk', 'Wine', 'Coke']
```

IF YOU INSERT MORE ITEMS THAN YOU REPLACE, THE NEW ITEMS WILL BE INSERTED WHERE YOU SPECIFIED, AND THE REMAINING ITEMS WILL MOVE ACCORDINGLY:

- **Example**
- Change the second value by replacing it with *two* new values:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      thislist[1:3] = ["wine"]  
      print(thislist)  
      ↗ ['Tea', 'wine']
```

Insert Items

To insert a new list item, without replacing any of the existing values, we can use the insert() method. The insert() method inserts an item at the specified index:

Example

Insert "wine" as the third item:

```
✓ 0s ▶ hislist = ["Tea", "Water", "Coffee"]  
      thislist.insert(2, "wine")  
      print(thislist)  
      ↗ ['Tea', 'wine', 'wine']
```

PYTHON - ADD LIST ITEMS

- **Append Items**

- To add an item to the end of the list, use the `append()` method:

- **Example**

- Using the `append()` method to append an item:

```
▶ thislist = ["Tea", "Water", "Coffee"]  
  thislist.append("milk")  
  print(thislist)  
→ ['Tea', 'Water', 'Coffee', 'milk']
```

Insert Items

To insert a list item at a specified index, use the `insert()` method.
The `insert()` method inserts an item at the specified index:

Example

Insert an item as the second position:

```
✓ 0s ▶ pplthislist = ["Tea", "Water", "Coffee"]  
      thislist.insert(1, "milk")  
      print(thislist)  
→ ['Tea', 'milk', 'Water', 'Coffee', 'milk']
```

Extend List

To append elements from *another list* to the current list, use the extend() method.

Example

Add the elements of tropical to thislist:

```
✓ 8 ▶ thislist = ["Tea", "Water", "Coffee"]  
    tropical = ["Juice", "Milk", "Wine"]  
    thislist.extend(tropical)  
    print(thislist)  
📄 ['Tea', 'Water', 'Coffee', 'Juice', 'Milk', 'Wine']
```

The elements will be added to the *end* of the list.

Add Any Iterable

The extend() method does not have to append *lists*, you can add any iterable object (tuples, sets, dictionaries etc.).

Example

Add elements of a tuple to a list:

```
✓ 8 ▶ thislist = ["Tea", "Water", "Coffee"]  
    thistuple = ("Juice", "Milk")  
    thislist.extend(thistuple)  
    print(thislist)  
📄 ['Tea', 'Water', 'Coffee', 'Juice', 'Milk']
```


PYTHON - REMOVE LIST ITEMS

Remove Specified Item

The `remove()` method removes the specified item.

Example

Remove "water":

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      thislist.remove("Water")  
      print(thislist)  
  
      ['Tea', 'Coffee']
```

Remove Specified Index

The `pop()` method removes the specified index.

Example

Remove the second item:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      thislist.pop(1)  
      print(thislist)  
  
      ['Tea', 'Coffee']
```


If you do not specify the index, the pop() method removes the last item.

- **Example**
- Remove the last item:

The del keyword also removes the specified index:

Example

Remove the first item:

```
▶ thislist = ["Tea", "Water", "Coffee"]  
  thislist.pop()  
  print(thislist)  
↳ ['Tea', 'Water']
```

```
▶ thislist = ["Tea", "Water", "Coffee"]  
  del thislist[0]  
  print(thislist)  
↳ ['Water', 'Coffee']
```

The del keyword can also delete the list completely.

- **Example**

- Delete the entire list:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      del thislist
```

Clear the List

The clear() method empties the list.

The list still remains, but it has no content.

Example

Clear the list content:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      thislist.clear()  
      print(thislist)  
  
[]
```

PYTHON - LOOP LISTS

Loop Through a List

You can loop through the list items by using a for loop:

Example

Print all items in the list, one by one:

```
▶ thislist = ["Tea", "Water", "Coffee"]  
  for x in thislist:  
    print(x)
```

↳ Tea
Water
Coffee

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the range() and len() functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
▶ thislist = ["Tea", "Water", "Coffee"]  
  for i in range(len(thislist)):  
    print(thislist[i])
```

↳ Tea
Water
Coffee

Using a While Loop

You can loop through the list items by using a while loop.

Use the len() function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes. Remember to increase the index by 1 after each iteration.

Example

Print all items, using a while loop to go through all the index numbers

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]
      i = 0
      while i < len(thislist):
          print(thislist[i])
          i = i + 1
```

```
☞ Tea
   Water
   Coffee
```

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists:

Example

A short hand for loop that will print all items in a list:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]
      [print(x) for x in thislist]
```

```
☞ Tea
   Water
   Coffee
   [None, None, None]
```

PYTHON - LIST COMPREHENSION

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of drinks, you want a new list, containing only the drinks with the letter "a" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:

Example

```
drinks = ["Tea", "Water", "Coffee", "Wine", "Milk"]
newlist = []

for x in drinks:
    if "a" in x:
        newlist.append(x)

print(newlist)
```

['Tea', 'Water']

With list comprehension you can do all that with only one line of code:

Example

```
drinks = ["Tea", "Water", "Coffee", "Wine", "Milk"]

newlist = [x for x in drinks if "a" in x]

print(newlist)
```

['Tea', 'Water']

THE SYNTAX

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The *condition* is like a filter that only accepts the items that evaluate to True.

Example

Only accept items that are not “tea”:

```
newlist = [x for x in drinks if x != "tea"]
```

Example

With no if statement:

```
newlist = [x for x in drinks]
```

ITERABLE

THE *ITERABLE* CAN BE ANY ITERABLE OBJECT, LIKE A LIST, TUPLE, SET ETC.

Example

You can use the range() function to create an iterable:

```
✓ 0s ▶ newlist = [x for x in range(10)]
```

Example

Accept only numbers lower than 5:

```
✓ 0s ▶ newlist = [x for x in range(10) if x < 5]
```

Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example

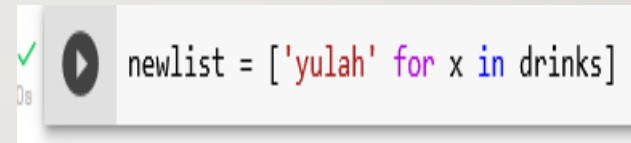
Set the values in the new list to upper case:

```
✓ 0s ▶ newlist = [x.upper() for x in drinks]
```

YOU CAN SET THE OUTCOME TO WHATEVER YOU LIKE:

- **Example**

- Set all values in the new list to yulah':



```
newlist = ['yulah' for x in drinks]
```

The *expression* can also contain conditions, not like a filter, but as a way to manipulate the outcome:

Example

Return “wine” instead of “water”:



```
newlist = [x if x != "water" else "wine" for x in drinks]
```

The *expression* in the example above says:

"Return the item if it is not water, if it is water return wine".

PYTHON - SORT LISTS

Sort List Alphanumerically

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

Example

Sort the list alphabetically:

```
▶ thislist = ["Tea", "Water", "Coffee", "Wine", "Milk"]  
  thislist.sort()  
  print(thislist)  
↳ ['Coffee', 'Milk', 'Tea', 'Water', 'Wine']
```

Example

Sort the list numerically:

```
▶ hislist = [100, 50, 65, 82, 23]  
  thislist.sort()  
  print(thislist)  
↳ ['Coffee', 'Milk', 'Tea', 'Water', 'Wine']
```

Sort Descending

To sort descending, use the keyword argument `reverse = True`:

- **Example**
- Sort the list descending:

```
▶ thislist = ["Tea", "Water", "Coffee", "Wine", "Milk"]  
  thislist.sort(reverse = True)  
  print(thislist)  
↳ ['Wine', 'Water', 'Tea', 'Milk', 'Coffee']
```

Example

Sort the list descending:

```
▶ thislist = [100, 50, 65, 82, 23]  
  thislist.sort(reverse = True)  
  print(thislist)  
↳ [100, 82, 65, 50, 23]
```

Customize Sort Function

You can also customize your own function by using the keyword argument `key = function`.

The function will return a number that will be used to sort the list (the lowest number first):

Example

Sort the list based on how close the number is to 50:

```
▶ def myfunc(n):  
  return abs(n - 50)  
  
  thislist = [100, 50, 65, 82, 23]  
  thislist.sort(key = myfunc)  
  print(thislist)  
↳ [50, 65, 23, 82, 100]
```


CASE INSENSITIVE SORT

BY DEFAULT THE SORT() METHOD IS CASE SENSITIVE, RESULTING IN ALL CAPITAL LETTERS BEING SORTED BEFORE LOWER CASE LETTERS:

- **Example**
- Case sensitive sorting can give an unexpected result:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Wine"]  
      thislist.sort()  
      print(thislist)  
  
      ['Coffee', 'Tea', 'Water', 'Wine']
```

Example

Perform a case-insensitive sort of the list:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Wine"]  
      thislist.sort(key = str.lower)  
      print(thislist)  
  
      ['Coffee', 'Tea', 'Water', 'Wine']
```

Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?

The reverse() method reverses the current sorting order of the elements.

Example

Reverse the order of the list items:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee", "Wine"]  
      thislist.reverse()  
      print(thislist)  
  
      ['Wine', 'Coffee', 'Water', 'Tea']
```

PYTHON - COPY LISTS

- Copy a List
- You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a reference to `list1`, and changes made in `list1` will automatically also be made in `list2`.
- There are ways to make a copy, one way is to use the built-in List method `copy()`.

Example

Make a copy of a list with the `copy()` method:

```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      mylist = thislist.copy()  
      print(mylist)  
  
      ['Tea', 'Water', 'Coffee']
```

Another way to make a copy is to use the built-in method `list()`.

Example

Make a copy of a list with the `list()` method:

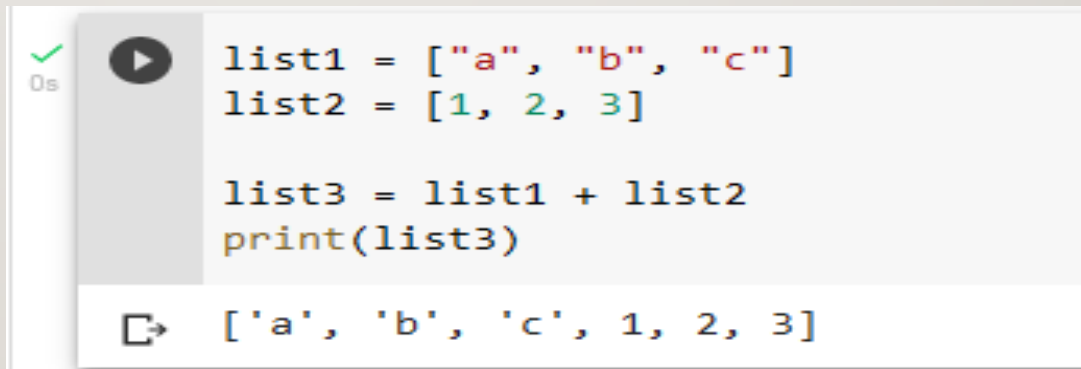
```
✓ 0s ▶ thislist = ["Tea", "Water", "Coffee"]  
      mylist = list(thislist)  
      print(mylist)  
  
      ['Tea', 'Water', 'Coffee']
```

PYTHON - JOIN LISTS

- Join Two Lists
- There are several ways to join, or concatenate, two or more lists in Python.
- One of the easiest ways are by using the + operator.

Example

Join two list:

A code execution snippet from a Python IDE. It shows a green checkmark and '0s' in the top left corner. The code defines two lists: list1 = ["a", "b", "c"] and list2 = [1, 2, 3]. Then it concatenates them into list3 = list1 + list2 and prints list3. The output is ['a', 'b', 'c', 1, 2, 3].

```
list1 = ["a", "b", "c"]  
list2 = [1, 2, 3]  
  
list3 = list1 + list2  
print(list3)
```

['a', 'b', 'c', 1, 2, 3]

ANOTHER WAY TO JOIN TWO LISTS IS BY APPENDING ALL THE ITEMS FROM LIST2 INTO LIST1, ONE BY ONE:

- **Example**
- Append list2 into list1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

for x in list2:
    list1.append(x)

print(list1)
```

['a', 'b', 'c', 1, 2, 3]

Or you can use the extend() method, which purpose is to add elements from one list to another list:

Example

Use the extend() method to add list2 at the end of list1:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

['a', 'b', 'c', 1, 2, 3]

PYTHON - LIST METHODS

- List Methods
- Python has a set of built-in methods that you can use on lists.

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

PYTHON TUPLES

```
mytuple = ("apple", "banana", "cherry")
```

Tuple

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple)
```

```
('apple', 'banana', 'cherry')
```

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Example

Tuples allow duplicate values:

```
✓ 0s ▶ thistuple = ("apple", "banana", "cherry", "apple", "cherry")  
      print(thistuple)  
  
📄 ('apple', 'banana', 'cherry', 'apple', 'cherry')
```

TUPLE LENGTH

TO DETERMINE HOW MANY ITEMS A TUPLE HAS, USE THE LEN() FUNCTION:

- Example
- Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

3

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example

One item tuple, remember the comma:

```
thistuple = ("apple",)
print(type(thistuple))

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

<class 'tuple'>
<class 'str'>

TUPLE ITEMS - DATA TYPES

TUPLE ITEMS CAN BE OF ANY DATA TYPE:

- Example
- String, int and boolean data types:

```
✓ 0s ▶ tuple1 = ("apple", "banana", "cherry")  
tuple2 = (1, 5, 7, 9, 3)  
tuple3 = (True, False, False)
```

A tuple can contain different data types:

Example

A tuple with strings, integers and boolean values:

```
✓ 0s ▶ tuple1 = ("abc", 34, True, 40, "female")
```

type()

From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
<class 'tuple'>
```


EXAMPLE

WHAT IS THE DATA TYPE OF A TUPLE?

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

<class 'tuple'>

The tuple() Constructor

It is also possible to use the tuple() constructor to make a tuple.

Example

Using the tuple() method to make a tuple:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

('apple', 'banana', 'cherry')

PYTHON - ACCESS TUPLE ITEMS

- **Access Tuple Items**
- **You can access tuple items by referring to the index number, inside square brackets:**

Example

Print the second item in the tuple:

```
✓ 0s ▶ thistuple = ("apple", "banana", "cherry")  
      print(thistuple[1])  
      banana
```

Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

Example

Print the last item of the tuple:

```
✓ 0s ▶ thistuple = ("apple", "banana", "cherry")  
      print(thistuple[-1])  
      cherry
```

RANGE OF INDEXES

YOU CAN SPECIFY A RANGE OF INDEXES BY SPECIFYING WHERE TO START AND WHERE TO END THE RANGE.

WHEN SPECIFYING A RANGE, THE RETURN VALUE WILL BE A NEW TUPLE WITH THE SPECIFIED ITEMS.

- **Example**
- Return the third, fourth, and fifth item:

```
✓ 0s ▶ thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
      print(thistuple[2:5])

      ('cherry', 'orange', 'kiwi')
```

By leaving out the start value, the range will start at the first item:

Example

This example returns the items from the beginning to, but NOT included, "kiwi":

```
✓ 2s ▶ thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
      print(thistuple[:4])

      ('apple', 'banana', 'cherry', 'orange')
```

BY LEAVING OUT THE END VALUE, THE RANGE WILL GO ON TO THE END OF THE LIST:

- Example
- This example returns the items from "cherry" and to the end:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[2:])  
  
( 'cherry', 'orange', 'kiwi', 'melon', 'mango')
```

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

Example

This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[-4:-1])  
  
( 'orange', 'kiwi', 'melon')
```

Check if Item Exists

To determine if a specified item is present in a tuple use the in keyword:

- **Example**
- Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")  
if "apple" in thistuple:  
    print("Yes, 'apple' is in the fruits tuple")
```

Yes, 'apple' is in the fruits tuple

PYTHON - UPDATE TUPLES

- Change Tuple Values
- Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.
- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Convert the tuple into a list to be able to change it:

```
✓ 0s ▶ x = ("apple", "banana", "cherry")
    y = list(x)
    y[1] = "kiwi"
    x = tuple(y)

    print(x)

↳ ('apple', 'kiwi', 'cherry')
```


ADD ITEMS

SINCE TUPLES ARE IMMUTABLE, THEY DO NOT HAVE A BUILD-IN APPEND() METHOD, BUT THERE ARE OTHER WAYS TO ADD ITEMS TO A TUPLE.

1. CONVERT INTO A LIST: JUST LIKE THE WORKAROUND FOR CHANGING A TUPLE, YOU CAN CONVERT IT INTO A LIST, ADD YOUR ITEM(S), AND CONVERT IT BACK INTO A TUPLE.

- **Example**

- Convert the tuple into a list, add "orange", and convert it back into a tuple:

2. Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Example

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

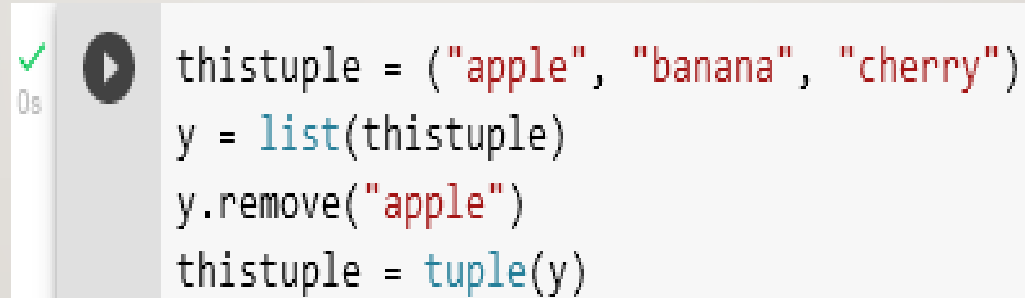
print(thistuple)
```

```
('apple', 'banana', 'cherry', 'orange')
```

REMOVE ITEMS

TUPLES ARE UNCHANGEABLE, SO YOU CANNOT REMOVE ITEMS FROM IT, BUT YOU CAN USE THE SAME WORKAROUND AS WE USED FOR CHANGING AND ADDING TUPLE ITEMS:

- **Example**
- Convert the tuple into a list, remove "apple", and convert it back into a tuple:

A code editor snippet with a green checkmark and a play button icon on the left. The code demonstrates how to remove an item from a tuple by converting it to a list, removing the item, and converting it back to a tuple.

```
thistuple = ("apple", "banana", "cherry")  
y = list(thistuple)  
y.remove("apple")  
thistuple = tuple(y)
```

OR YOU CAN DELETE THE TUPLE COMPLETELY:

Example

The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

```
NameError                                Traceback (most recent call last)
<ipython-input-99-5ea6b628c556> in <module>
      1 thistuple = ("apple", "banana", "cherry")
      2 del thistuple
----> 3 print(thistuple) #this will raise an error because the tuple no longer exists

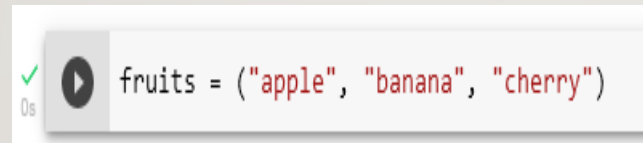
NameError: name 'thistuple' is not defined
```

SEARCH STACK OVERFLOW

PYTHON - UNPACK TUPLES

- Unpacking a Tuple
- When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

Example

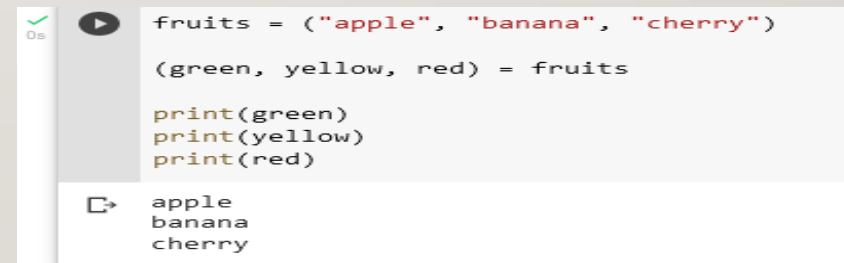
A code editor snippet with a green checkmark and a play button icon. It contains the Python code: `fruits = ("apple", "banana", "cherry")`

```
fruits = ("apple", "banana", "cherry")
```

Packing a tuple:

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Example
Unpacking a tuple:

A code editor snippet with a green checkmark and a play button icon. It contains the Python code for unpacking a tuple. Below the code, the output of the print statements is shown: 'apple', 'banana', and 'cherry' on separate lines.

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

apple
banana
cherry

USING ASTERISK*

IF THE NUMBER OF VARIABLES IS LESS THAN THE NUMBER OF VALUES, YOU CAN ADD AN * TO THE VARIABLE NAME AND THE VALUES WILL BE ASSIGNED TO THE VARIABLE AS A LIST:

- **Example**
- Assign the rest of the values as a list called "red":

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Example

Add a list of values the "tropic" variable:

```
✓ 0s ▶ fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
    (green, yellow, *red) = fruits

    print(green)
    print(yellow)
    print(red)
```

```
📄 apple
   banana
   ['cherry', 'strawberry', 'raspberry']
```

```
✓ 0s ▶ fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
    (green, *tropic, red) = fruits

    print(green)
    print(tropic)
    print(red)
```

```
📄 apple
   ['mango', 'papaya', 'pineapple']
   cherry
```


Python - Loop Tuples

Loop Through a Tuple

You can loop through the tuple items by using a for loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")  
for x in thistuple:  
    print(x)
```

```
apple  
banana  
cherry
```

Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number.

Example

Print all items by referring to their index number:

Use the range() and len() functions to create a suitable iterable.

```
thistuple = ("apple", "banana", "cherry")  
for i in range(len(thistuple)):  
    print(thistuple[i])
```

```
apple  
banana  
cherry
```

Using a While Loop

You can loop through the list items by using a while loop.

Use the len() function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

Example

Print all items, using a while loop to go through all the index numbers:



```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
    print(thistuple[i])
    i = i + 1
```

apple
banana
cherry

Learn more about while loops in our [Python While Loops Chapter](#).

PYTHON - JOINTUPLES

Join Two Tuples

To join two or more tuples you can use the + operator:

Example

Join two tuples:

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

```
('a', 'b', 'c', 1, 2, 3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Example

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```


```
('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

PYTHON - TUPLE METHODS

- **Tuple Methods**
- Python has two built-in methods that you can use on tuples.

Method	Description
<code>count()</code>	Returns the number of times a specified value occurs in a tuple
<code>index()</code>	Searches the tuple for a specified value and returns the position of where it was found

PYTHON SETS

```
✓  myset = {"apple", "banana", "cherry"}
```

- Set
- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is unordered, unchangeable*, and unindexed.

Sets are written with curly brackets.

Example
Create a Set:

```
✓  thisset = {"apple", "banana", "cherry"}  
print(thisset)  
  
{'banana', 'cherry', 'apple'}
```


SET ITEMS

SET ITEMS ARE UNORDERED, UNCHANGEABLE, AND DO NOT ALLOW DUPLICATE VALUES.

- **Unordered**

- Unordered means that the items in a set do not have a defined order.
- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

Duplicates Not Allowed

Sets cannot have two items with the same value.



EXAMPLE

DUPLICATE VALUES WILL BE IGNORED:

```
✓ 0s ▶ thisset = {"apple", "banana", "cherry", "apple"}  
      print(thisset)  
  
      {'banana', 'cherry', 'apple'}
```

Get the Length of a Set

To determine how many items a set has, use the len() function.

Example

Get the number of items in a set:

```
✓ 0s ▶ thisset = {"apple", "banana", "cherry"}  
      print(len(thisset))  
  
      3
```

SET ITEMS - DATA TYPES

SET ITEMS CAN BE OF ANY DATA TYPE:

- **Example**
- String, int and boolean data types:

```
✓ 0s ▶ set1 = {"apple", "banana", "cherry"}  
      set2 = {1, 5, 7, 9, 3}  
      set3 = {True, False, False}
```

A set can contain different data types:

Example

A set with strings, integers and boolean values:

```
✓ 0s ▶ set1 = {"abc", 34, True, 40, "female"}
```

type()

From Python's perspective, sets are defined as objects with the data type 'set':

```
<class 'set'>
```

EXAMPLE

WHAT IS THE DATA TYPE OF A SET?

```
✓ 0s ▶ myset = {"apple", "banana", "cherry"}  
print(type(myset))  
  
<class 'set'>
```

The set() Constructor

It is also possible to use the set() constructor to make a set.

Example

Using the set() constructor to make a set:

```
✓ 0s ▶ thisset = set(("apple", "banana", "cherry")) # note the double round-brackets  
print(thisset)  
  
{'banana', 'cherry', 'apple'}
```

PYTHON - ACCESS SET ITEMS

Access Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)
```

banana
cherry
apple

Example

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}  
  
print("banana" in thisset)
```

True

PYTHON - ADD SET ITEMS

ADD ITEMS

Example

Add an item to a set, using the add() method:

Add Sets

To add items from another set into the current set, use the update() method.

Example

Add elements from tropical into thisset:

```
✓ 0s ▶ thisset = {"apple", "banana", "cherry"}
      thisset.add("orange")
      print(thisset)

{'banana', 'cherry', 'orange', 'apple'}
```

```
✓ 0s ▶ thisset = {"apple", "banana", "cherry"}
      tropical = {"pineapple", "mango", "papaya"}

      thisset.update(tropical)
      print(thisset)

📄 {'banana', 'mango', 'cherry', 'papaya', 'pineapple', 'apple'}
```

ADD ANY ITERABLE

THE OBJECT IN THE UPDATE() METHOD DOES NOT HAVE TO BE A SET, IT CAN BE ANY ITERABLE OBJECT (TUPLES, LISTS, DICTIONARIES ETC.).

- **Example**
- Add elements of a list to a set:

```
✓ 0s ▶ thisset = {"apple", "banana", "cherry"}  
      mylist = ["kiwi", "orange"]  
  
      thisset.update(mylist)  
  
      print(thisset)  
  
↗ {'apple', 'banana', 'kiwi', 'cherry', 'orange'}
```



PYTHON - REMOVE SET ITEMS

REMOVE ITEM

TO REMOVE AN ITEM IN A SET, USE THE REMOVE(), OR THE DISCARD() METHOD.



Example

Remove "banana" by using the remove() method:

```
0s  thisset = {"apple", "banana", "cherry"}  
thisset.remove("banana")  
print(thisset)  
 {'cherry', 'apple'}
```

Example

Remove "banana" by using the discard() method:

```
1s  thisset = {"apple", "banana", "cherry"}  
thisset.discard("banana")  
print(thisset)  
 {'cherry', 'apple'}
```

YOU CAN ALSO USE THE POP() METHOD TO REMOVE AN ITEM, BUT THIS METHOD WILL REMOVE THE LAST ITEM. REMEMBER THAT SETS ARE UNORDERED, SO YOU WILL NOT KNOW WHAT ITEM THAT GETS REMOVED.

THE RETURNVALUE OF THE POP() METHOD IS THE REMOVED ITEM.

Example

Remove the last item by using the pop() method:

```
✓ 0s ▶ thisset = {"apple", "banana", "cherry"}  
      x = thisset.pop()  
      print(x)  
      print(thisset)  
      banana  
      {'cherry', 'apple'}
```

Example

The clear() method empties the set:

```
✓ 0s ▶ thisset = {"apple", "banana", "cherry"}  
      thisset.clear()  
      print(thisset)  
      set()
```

PYTHON - LOOP SETS

- Loop Items
- You can loop through the set items by using a for loop:

Example

Loop through the set, and print the values:



```
✓ 0s ▶ thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)  
  
banana  
cherry  
apple
```

The image shows a code execution environment. On the left, there is a green checkmark and '0s' indicating successful execution. A play button icon is next to the code. The code defines a set 'thisset' with elements 'apple', 'banana', and 'cherry'. It then uses a 'for' loop to iterate over the set and print each item. The output shows the items 'banana', 'cherry', and 'apple' printed on separate lines.

PYTHON - JOIN SETS JOIN TWO SETS

THERE ARE SEVERAL WAYS TO JOIN TWO OR MORE SETS IN PYTHON.

YOU CAN USE THE UNION() METHOD THAT RETURNS A NEW SET CONTAINING ALL ITEMS FROM BOTH SETS, OR THE UPDATE() METHOD THAT INSERTS ALL THE ITEMS FROM ONE SET INTO ANOTHER:

- Example
- The union() method returns a new set with all items from both sets:

```
✓ 0s ▶ set1 = {"a", "b", "c"}  
      set2 = {1, 2, 3}  
  
      set3 = set1.union(set2)  
      print(set3)  
  
📄 {1, 2, 3, 'b', 'a', 'c'}
```

Example

The update() method inserts the items in set2 into set1:

```
✓ 0s ▶ set1 = {"a", "b", "c"}  
      set2 = {1, 2, 3}  
  
      set1.update(set2)  
      print(set1)  
  
📄 {1, 2, 3, 'b', 'a', 'c'}
```

KEEP ONLY THE DUPLICATES

THE INTERSECTION_UPDATE() METHOD WILL KEEP ONLY THE ITEMS THAT ARE PRESENT IN BOTH SETS.

- Example
- Keep the items that exist in both set x, and set y:

Example

Return a set that contains the items that exist in both set x, and set y:

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
x.intersection_update(y)  
  
print(x)  
  
{'apple'}
```

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
z = x.intersection(y)  
  
print(z)  
  
{'apple'}
```

KEEP ALL, BUT NOT THE DUPLICATES

THE SYMMETRIC_DIFFERENCE_UPDATE() METHOD WILL KEEP ONLY THE ELEMENTS THAT ARE NOT PRESENT IN BOTH SETS.

- Example
- Keep the items that are not present in both sets:

Example

Return a set that contains all items from both sets, except items that are present in both:

```
✓ 0s ▶ x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
x.symmetric_difference_update(y)  
  
print(x)  
  
{'banana', 'google', 'microsoft', 'cherry'}
```

```
✓ 0s ▶ x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
z = x.symmetric_difference(y)  
  
print(z)  
  
▶ {'banana', 'google', 'microsoft', 'cherry'}
```

PYTHON - SET METHODS

SET METHODS

PYTHON HAS A SET OF BUILT-IN METHODS THAT YOU CAN USE ON SETS.

Method	Description
<u>add()</u>	Adds an element to the set
<u>clear()</u>	Removes all the elements from the set
<u>copy()</u>	Returns a copy of the set
<u>difference()</u>	Returns a set containing the difference between two or more sets
<u>difference_update()</u>	Removes the items in this set that are also included in another, specified set
<u>discard()</u>	Remove the specified item
<u>intersection()</u>	Returns a set, that is the intersection of two other sets
<u>intersection_update()</u>	Removes the items in this set that are not present in other, specified set(s)
<u>isdisjoint()</u>	Returns whether two sets have a intersection or not
<u>issubset()</u>	Returns whether another set contains this set or not
<u>issuperset()</u>	Returns whether this set contains another set or not
<u>pop()</u>	Removes an element from the set
<u>remove()</u>	Removes the specified element
<u>symmetric_difference()</u>	Returns a set with the symmetric differences of two sets
<u>symmetric_difference_update()</u>	inserts the symmetric differences from this set and another
<u>union()</u>	Return a set containing the union of sets
<u>update()</u>	Update the set with the union of this set and others

PYTHON DICTIONARIES

- **Dictionary**
- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values:
- **Example**
- Create and print a dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict)
```


DICTIONARY ITEMS

DICTIONARY ITEMS ARE ORDERED, CHANGEABLE, AND DOES NOT ALLOW DUPLICATES. DICTIONARY ITEMS ARE PRESENTED IN KEY:VALUE PAIRS, AND CAN BE REFERRED TO BY USING THE KEY NAME.

- **Example**
- Print the "brand" value of the dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict["brand"])
```

Ordered or Unordered?

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

DUPLICATES NOT ALLOWED

DICTIONARIES CANNOT HAVE TWO ITEMS WITH THE SAME KEY:

- **Example**
- Duplicate values will overwrite existing values:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```

PYTHON - ACCESS DICTIONARY ITEMS

ACCESSING ITEMS

YOU CAN ACCESS THE ITEMS OF A DICTIONARY BY REFERRING TO ITS KEY NAME, INSIDE SQUARE BRACKETS:

- **Example**
- Get the value of the "model" key:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
x = thisdict["model"]
```

PYTHON - CHANGE DICTIONARY ITEMS CHANGE VALUES

YOU CAN CHANGE THE VALUE OF A SPECIFIC ITEM BY REFERRING TO ITS KEY NAME:

- **Example**
- Change the "year" to 2018:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["year"] = 2018
```

Update Dictionary

The update() method will update the dictionary with the items from the given argument.
The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the update() method:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"year": 2020})
```

PYTHON - ADD DICTIONARY ITEMS

ADDING ITEMS

ADDING AN ITEM TO THE DICTIONARY IS DONE BY USING A NEW INDEX KEY AND ASSIGNING A VALUE TO IT:

- **Example**

Update Dictionary

The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Add a color item to the dictionary by using the update() method:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict["color"] = "red"  
print(thisdict)
```

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.update({"color": "red"})
```


PYTHON - REMOVE DICTIONARY ITEMS

REMOVING ITEMS

THERE ARE SEVERAL METHODS TO REMOVE ITEMS FROM A DICTIONARY:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.pop("model")  
print(thisdict)
```

Example

The `popitem()` method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
thisdict.popitem()  
print(thisdict)
```

PYTHON - LOOP DICTIONARIES LOOP THROUGH A DICTIONARY

YOU CAN LOOP THROUGH A DICTIONARY BY USING A FOR LOOP.

WHEN LOOPING THROUGH A DICTIONARY, THE RETURN VALUE ARE THE KEYS OF THE DICTIONARY, BUT THERE ARE METHODS TO RETURN THE VALUES AS WELL.

- **Example**
- Print all key names in the dictionary, one by one:

```
for x in thisdict:  
    print(x)
```

Example

Print all *values* in the dictionary, one by one:

```
for x in thisdict:  
    print(thisdict[x])
```

PYTHON - COPY DICTIONARIES COPY A DICTIONARY

YOU CANNOT COPY A DICTIONARY SIMPLY BY TYPING `DICT2 = DICT1`, BECAUSE: `DICT2` WILL ONLY BE A REFERENCE TO `DICT1`, AND CHANGES MADE IN `DICT1` WILL AUTOMATICALLY ALSO BE MADE IN `DICT2`.

THERE ARE WAYS TO MAKE A COPY, ONE WAY IS TO USE THE BUILT-IN DICTIONARY METHOD `COPY()`.

Example

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = thisdict.copy()  
print(mydict)
```

Example

Make a copy of a dictionary with the `dict()` function:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
mydict = dict(thisdict)  
print(mydict)
```

PYTHON - NESTED DICTIONARIES

NESTED DICTIONARIES

A DICTIONARY CAN CONTAIN DICTIONARIES, THIS IS CALLED NESTED DICTIONARIES.

- **Example**
- Create a dictionary that contain three dictionaries:

```
0s ✓ myfamily = {  
    "child1" : {  
        "name" : "Emil",  
        "year" : 2004  
    },  
    "child2" : {  
        "name" : "Tobias",  
        "year" : 2007  
    },  
    "child3" : {  
        "name" : "Linus",  
        "year" : 2011  
    }  
}
```

Example

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
0s ✓ child1 = {  
    "name" : "Emil",  
    "year" : 2004  
}  
child2 = {  
    "name" : "Tobias",  
    "year" : 2007  
}  
child3 = {  
    "name" : "Linus",  
    "year" : 2011  
}  
  
myfamily = {  
    "child1" : child1,  
    "child2" : child2,  
    "child3" : child3  
}
```

PYTHON DICTIONARY METHODS

DICTIONARY METHODS

PYTHON HAS A SET OF BUILT-IN METHODS THAT YOU CAN USE ON DICTIONARIES.

Method	Description
<u>clear()</u>	Removes all the elements from the dictionary
<u>copy()</u>	Returns a copy of the dictionary
<u>fromkeys()</u>	Returns a dictionary with the specified keys and value
<u>get()</u>	Returns the value of the specified key
<u>items()</u>	Returns a list containing a tuple for each key value pair
<u>keys()</u>	Returns a list containing the dictionary's keys
<u>pop()</u>	Removes the element with the specified key
<u>popitem()</u>	Removes the last inserted key-value pair
<u>setdefault()</u>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<u>update()</u>	Updates the dictionary with the specified key-value pairs
<u>values()</u>	Returns a list of all the values in the dictionary

PYTHON IF ... ELSE

- **Python Conditions and If statements**
- Python supports the usual logical conditions from mathematics:
- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`
- These conditions can be used in several ways, most commonly in "if statements" and loops.
- An "if statement" is written by using the if keyword.

EXAMPLE

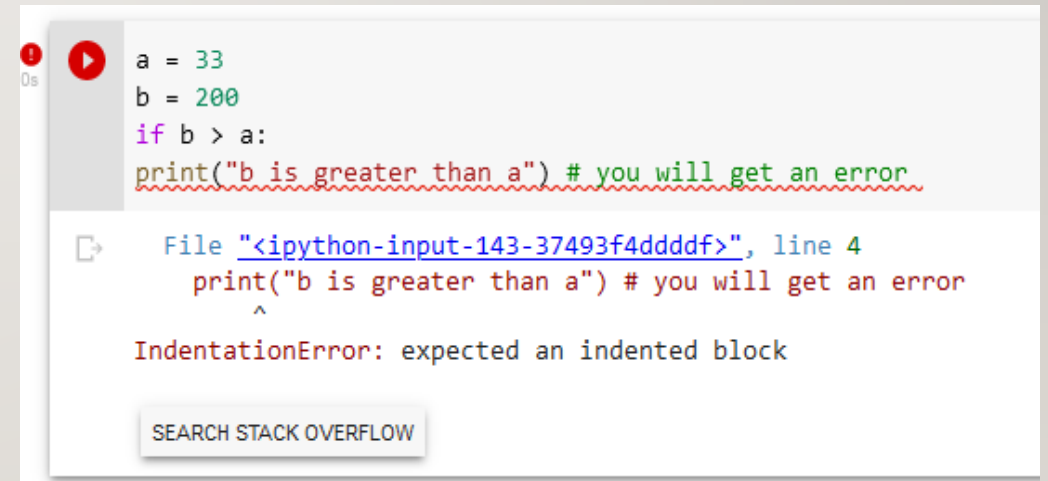
IF STATEMENT:

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

- In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".
- **Indentation**
- Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):



```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

File "<ipython-input-143-37493f4dddf>", line 4
 print("b is greater than a") # you will get an error
 ^
IndentationError: expected an indented block

SEARCH STACK OVERFLOW

PYTHON WHILE LOOPS

- Python Loops
- Python has two primitive loop commands:
 - while loops
 - for loops

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

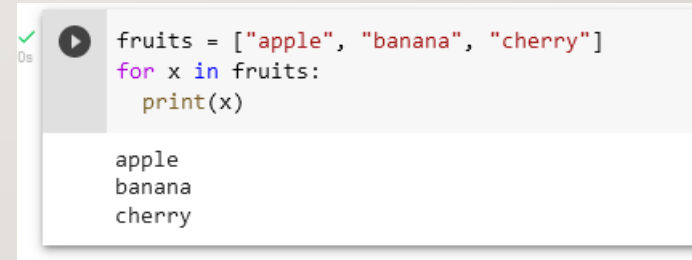
```
i = 1
while i < 6:
    print(i)
    i += 1
```

PYTHON FOR LOOPS

- Python For Loops
- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

Print each fruit in a fruit list:

A screenshot of a code editor showing a Python script. The script defines a list 'fruits' with elements 'apple', 'banana', and 'cherry'. It then uses a 'for' loop to iterate over each item 'x' in the list and prints it. The output of the script is displayed below the code, showing 'apple', 'banana', and 'cherry' on separate lines. The code is color-coded: 'fruits' is blue, 'apple', 'banana', and 'cherry' are in quotes and red, 'for' is blue, 'x' is purple, 'in' is blue, and 'print' is orange.

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

apple
banana
cherry

PYTHON FUNCTIONS

- Creating a Function
- In Python a function is defined using the def keywo
- **Example**

```
def my_function():  
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():  
    print("Hello from a function")  
  
my_function()
```


PYTHON LAMBDA



A LAMBDA FUNCTION IS A SMALL ANONYMOUS FUNCTION.

- A lambda function can take any number of arguments, but can only have one expression.
- **Syntax**
- lambda *arguments* : *expression*
- The expression is executed and the result is returned:

EXAMPLE

ADD 10 TO ARGUMENT A, AND RETURN THE RESULT:

```
x = lambda a : a + 10
print(x(5))
```

15

Lambda functions can take any number of arguments:

Example

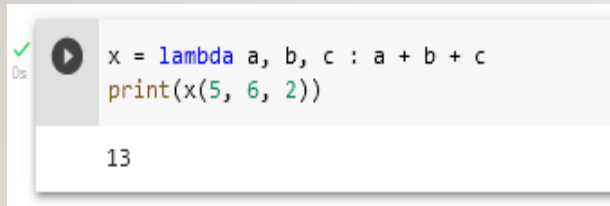
Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

30

EXAMPLE

SUMMARIZE ARGUMENT A, B, AND C AND RETURN THE RESULT:



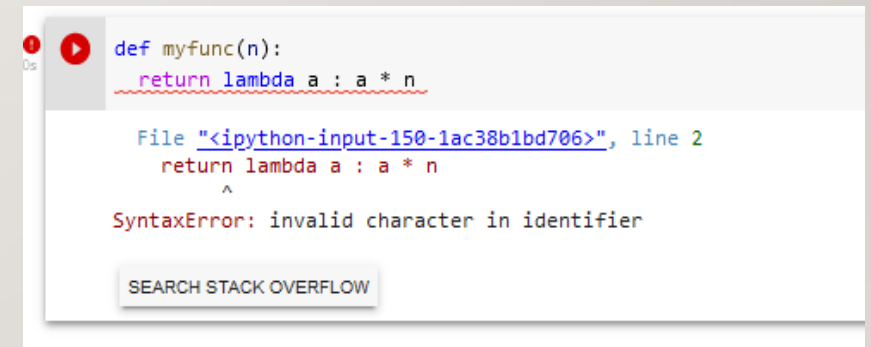
```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

13

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:



```
def myfunc(n):
    return lambda a : a * n
```

File "<ipython-input-150-1ac38b1bd706>", line 2
return lambda a : a * n
 ^
SyntaxError: invalid character in identifier

SEARCH STACK OVERFLOW

USE THAT FUNCTION DEFINITION TO MAKE A FUNCTION THAT ALWAYS DOUBLES THE NUMBER YOU SEND IN:

- Example

```
def myfunc(n):  
    return lambda a : a * n  
  
mydoubler = myfunc(2)  
  
print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

```
def myfunc(n):  
    return lambda a : a * n  
  
mytripler = myfunc(3)  
  
print(mytripler(11))
```


OR, USE THE SAME FUNCTION DEFINITION TO MAKE BOTH FUNCTIONS, IN THE SAME PROGRAM:

- **Example**

```
def myfunc(n):  
    return lambda a : a * n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11))  
print(mytripler(11))
```