

PYTHON

Welcome!

- This class is about more than computer programming!
- Indeed, this class is about problem-solving in a way that is exceedingly empowering! You will likely take the problem solving that you learn here will likely be instantly applicable to your work beyond this course and even your career as a whole!
- However, it will not be easy! You will be “drinking from the firehose” of knowledge during this course. You’ll be amazed at what you will be able to accomplish in the coming weeks.

PYTHON

Welcome!

- This course is far more about you advancing “you” from “where you are today” than hitting some imagined standard.
- The most important opening consideration in this course: Give the time you need to learn through this course. Everyone learns differently. If something does not work out well at the start, know that with time you will grow and grow in your skill.

PYTHON

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.

PYTHON

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.

PYTHON

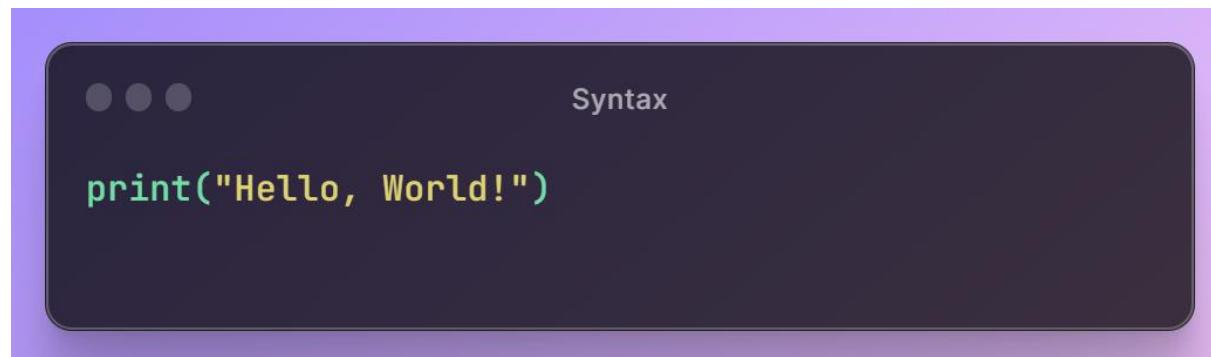
Why Python ?

- 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.

PYTHON

Python Syntax Compared to other programming languages

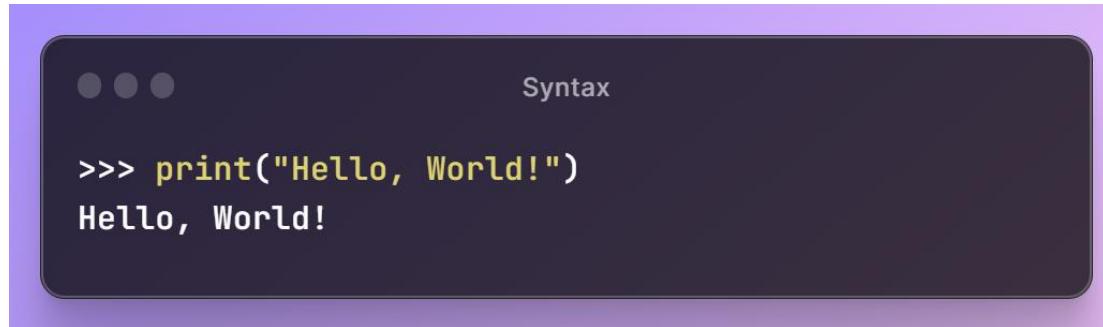
- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.



PYTHON

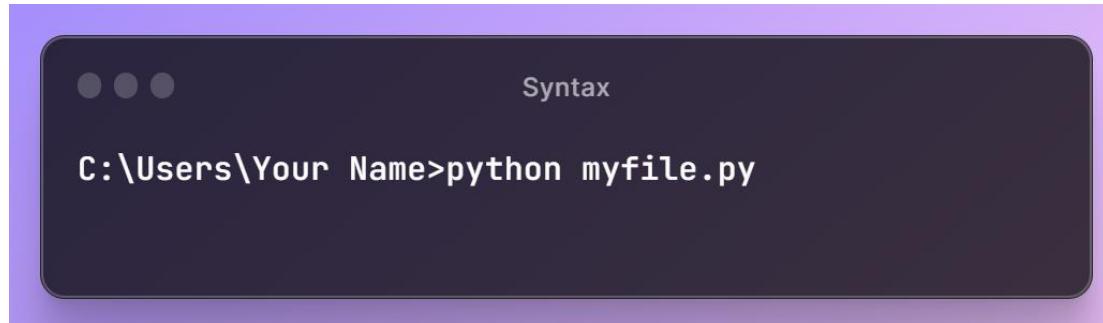
Execute Python Syntax

- As we learned in the previous page, Python syntax can be executed by writing directly in the Command Line:



A screenshot of a terminal window titled "Syntax". It shows the command `>>> print("Hello, World!")` being typed, followed by the output `Hello, World!`.

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



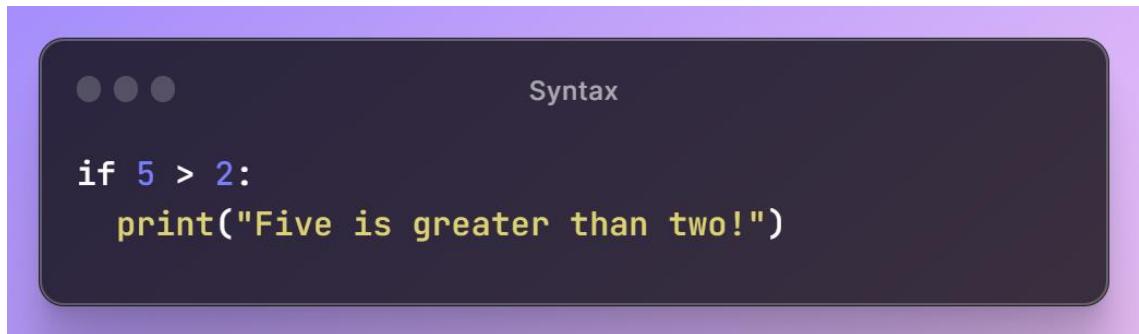
A screenshot of a terminal window titled "Syntax". It shows the command `C:\Users\Your Name>python myfile.py` being typed.

PYTHON

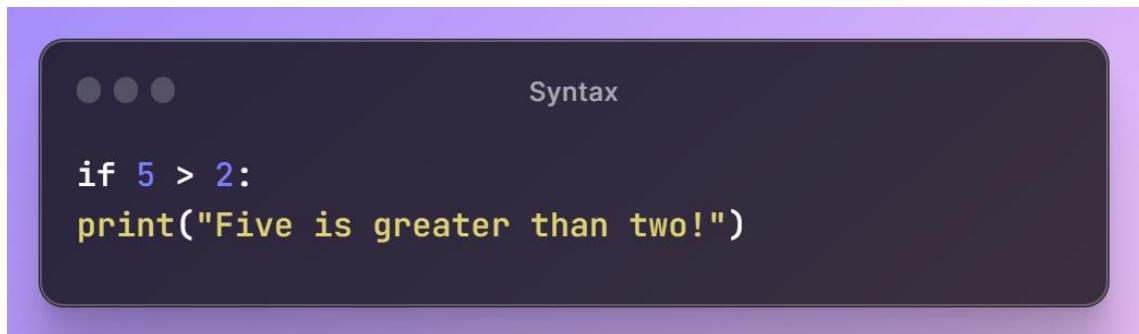
Python Indentation

- 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 will give you an error if you skip the indentation:



The screenshot shows a dark-themed code editor window. In the top right corner, the word "Syntax" is displayed above three small circular icons. The main area contains the following Python code:
`if 5 > 2:
 print("Five is greater than two!")`



The screenshot shows a dark-themed code editor window. In the top right corner, the word "Syntax" is displayed above three small circular icons. The main area contains the following Python code, which lacks indentation for the print statement:
`if 5 > 2:
print("Five is greater than two!")`

Python Indentation

- The number of spaces is up to you as a programmer, the most common use is four, but it has to be at least one.
- You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

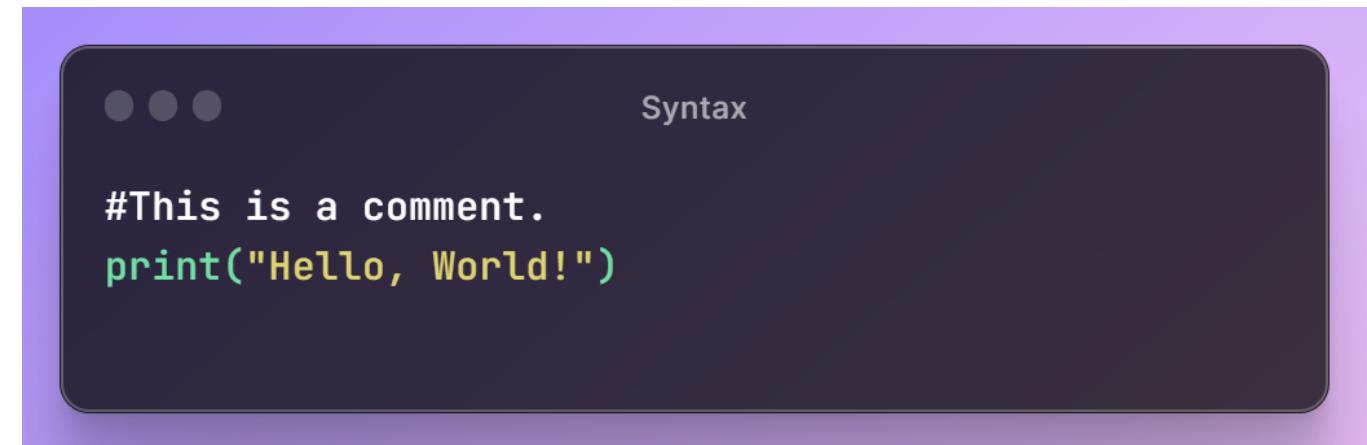
```
... Syntax  
if 5 > 2:  
    print("Five is greater than two!")  
if 5 > 2:  
    print("Five is greater than two!")
```

```
... Syntax  
if 5 > 2:  
    print("Five is greater than two!")  
        print("Five is greater than two!")
```

PYTHON

Comments

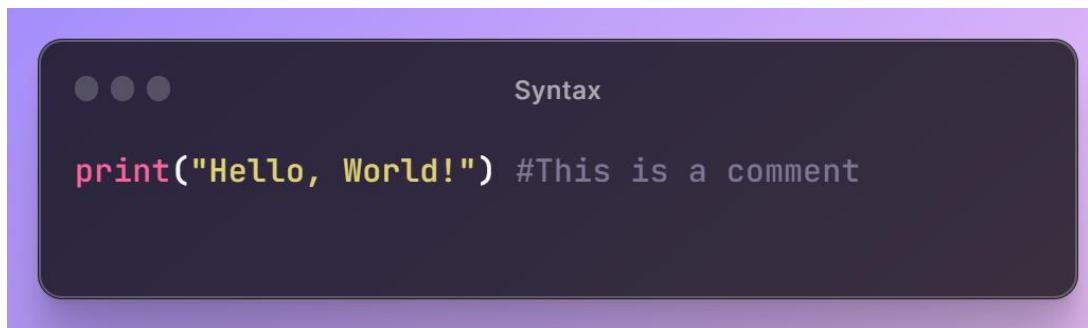
- Python has commenting capability for the purpose of in-code documentation.
- Comments start with “#”, and Python will render the rest of the line as a comment:
- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.



PYTHON

Comments

- Comments can be placed at the end of a line, and Python will ignore the rest of the line:

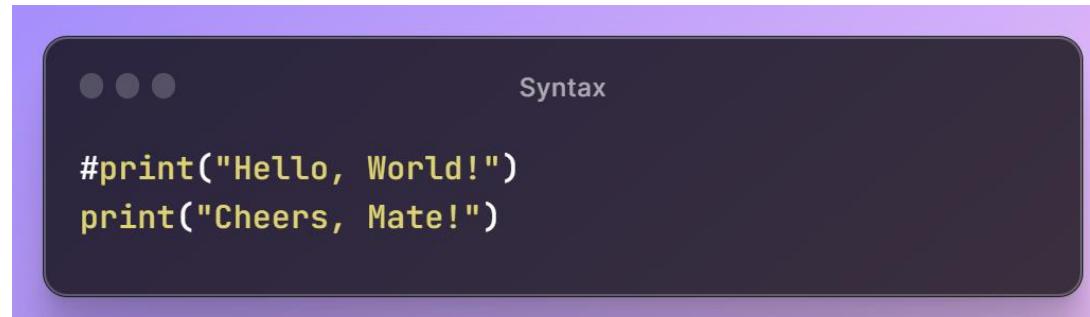


A screenshot of a Python code editor. The code shown is:

```
... Syntax  
print("Hello, World!") #This is a comment
```

The code is displayed in a dark-themed code editor with syntax highlighting. The comment is preceded by a hash symbol (#) and spans the rest of the line.

- A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code:



A screenshot of a Python code editor. The code shown is:

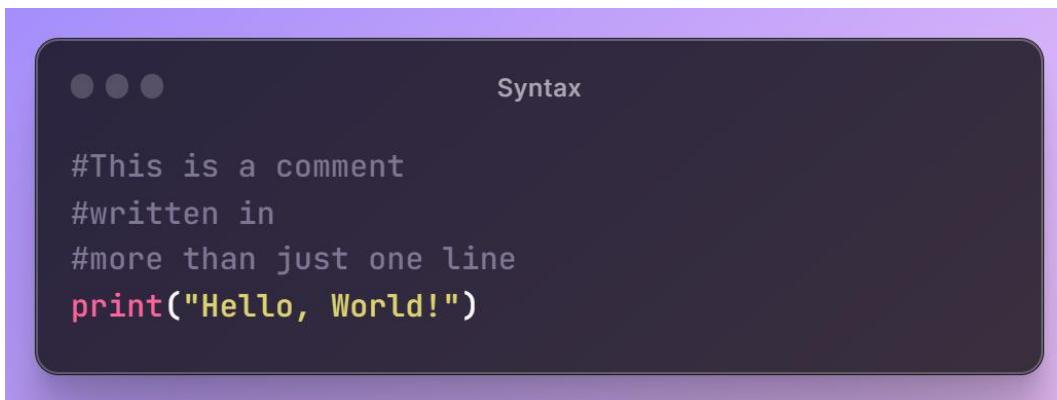
```
... Syntax  
#print("Hello, World!")  
print("Cheers, Mate!")
```

This demonstrates a multi-line comment, where the entire block of code is ignored by Python due to the leading hash symbols on each line.

PYTHON

Multiline Comments

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



The screenshot shows a dark-themed code editor window. In the top right corner, there is a small icon consisting of three dots. To its right, the word "Syntax" is displayed. Below this, a multi-line string is shown in yellow text:
`#This is a comment
#written in
#more than just one line
print("Hello, World!")`

- 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:



The screenshot shows a dark-themed code editor window. In the top right corner, there is a small icon consisting of three dots. To its right, the word "Syntax" is displayed. Below this, a multi-line string is shown in yellow text, enclosed in triple quotes:
`"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")`

- Or, not quite as intended, you can use a multiline string.

- 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.

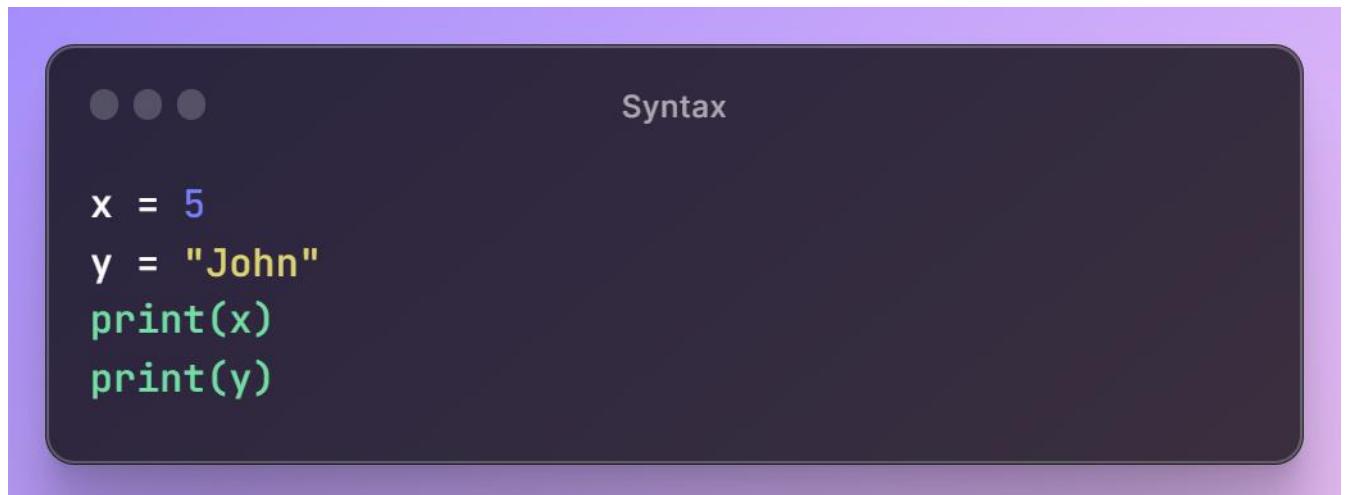
PYTHON

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.



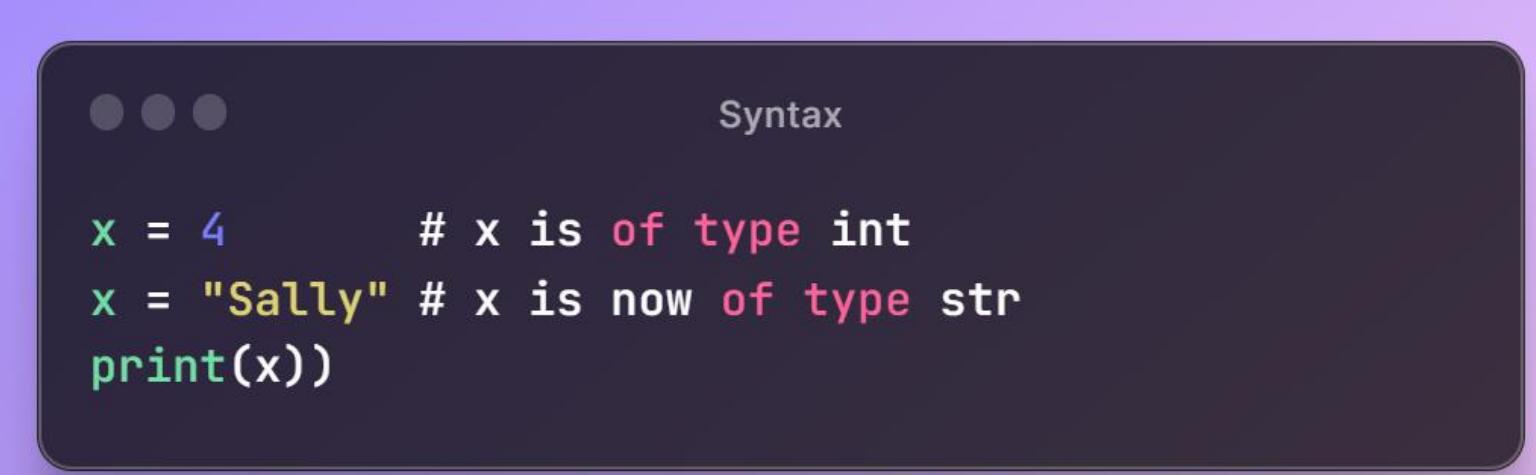
The image shows a dark-themed code editor window with a purple header bar. The header bar contains three small gray dots on the left and the word "Syntax" on the right. The main area of the editor is dark gray with white text. It displays the following Python code:

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

PYTHON

Variables

- Variables do not need to be declared with any particular *type*, and can even change type after they have been set.



The image shows a dark-themed code editor window with a light purple header bar. In the top right corner of the header bar, there are three small circular icons. To the right of these icons, the word "Syntax" is written in a light gray font. The main area of the code editor contains the following Python code:

```
x = 4      # x is of type int
x = "Sally" # x is now of type str
print(x))
```

PYTHON

Casting

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

```
• • • Syntax  
x = str(3)      # x will be '3'  
y = int(3)       # y will be 3  
z = float(3)     # z will be 3.0
```

Get the Type

- You can get the data type of a variable with the `type()` function.

```
• • • Syntax  
x = 5  
y = "John"  
print(type(x))  
print(type(y))
```

PYTHON

Single or Double Quotes?

- String variables can be declared either by using single or double quotes:

```
...  
Syntax  
  
x = "John"  
# is the same as  
x = 'John'
```

Case-Sensitivity

- Variable names are case-sensitive.

```
...  
Syntax  
  
This will create two variables:  
  
a = 4  
A = "Sally"  
#A will not overwrite a
```

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)
- A variable name cannot be any of the [Python keywords](#).

PYTHON

Variable Names



syntax

Legal variable names:

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



syntax

Illegal variable names:

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

Remember that variable names are case-sensitive

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:

```
myVariableName = "John"
```

Pascal Case

- Each word starts with a capital letter:

```
MyVariableName = "John"
```

Snake Case

- Each word is separated by an underscore character:

```
my_variable_name = "John"
```

Assign Multiple Values

- Many Values to Multiple Variables
- Python allows you to assign values to multiple variables in one line:

• • •

syntax

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

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

• • •

syntax

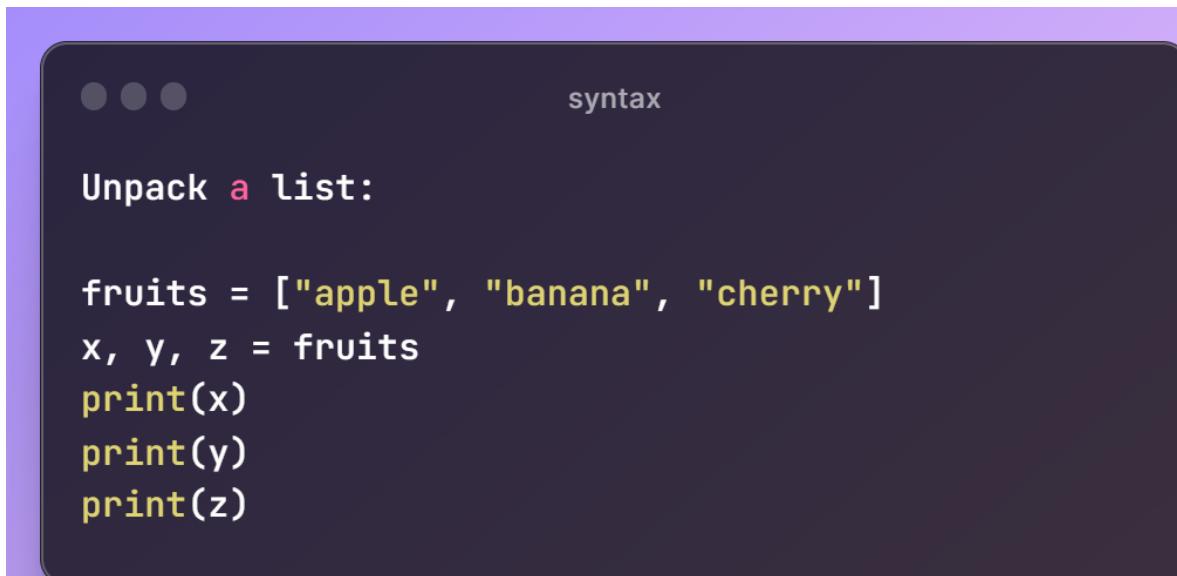
```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Note: Make sure the number of variables matches the number of values, or else you will get an error.

Assign Multiple Values

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*.



The image shows a dark-themed code editor window with a purple header bar. In the top right corner of the header bar, there are three small white dots. To the right of these dots, the word "syntax" is written in a light gray font. The main area of the code editor is dark gray with white text. At the top left of this area, there are three small white dots. Below these dots, the text "Unpack a list:" is displayed in a light gray font. Underneath this heading, there is a block of Python code:

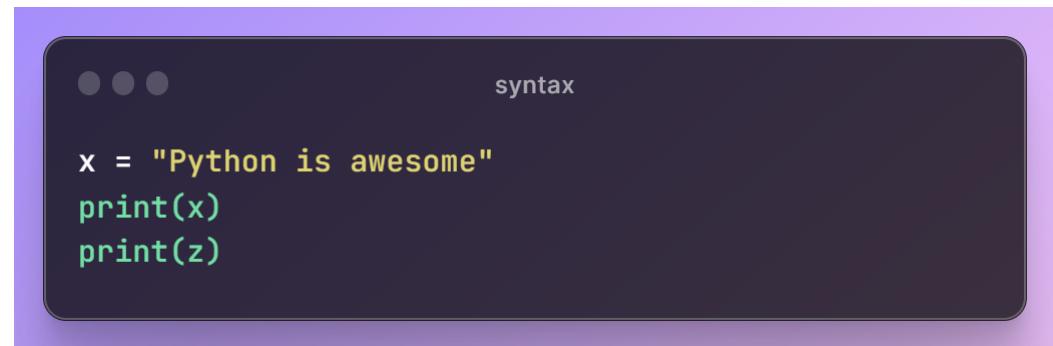
```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
print(x)
print(y)
print(z)
```

PYTHON

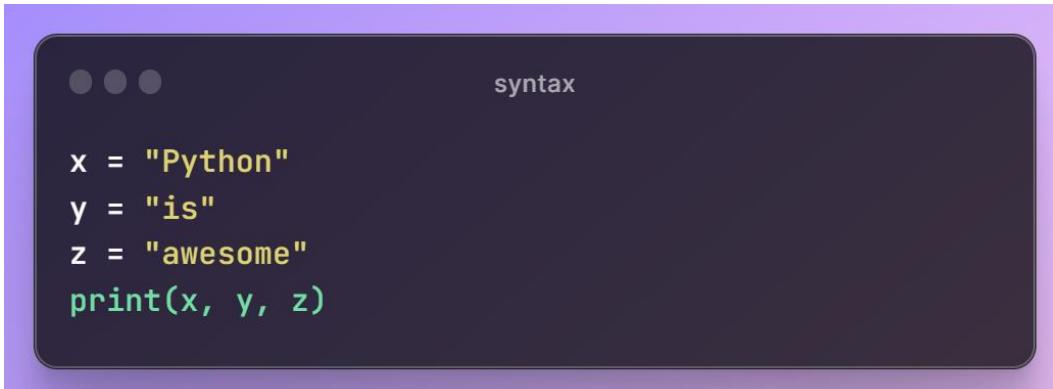
Output Variables

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

In the `print()` function, you output multiple variables, separated by a comma:



A screenshot of a code editor window with a dark theme. In the top right corner, the word "syntax" is displayed above three small circular icons. The main area contains the following Python code:
x = "Python is awesome"
print(x)
print(z)

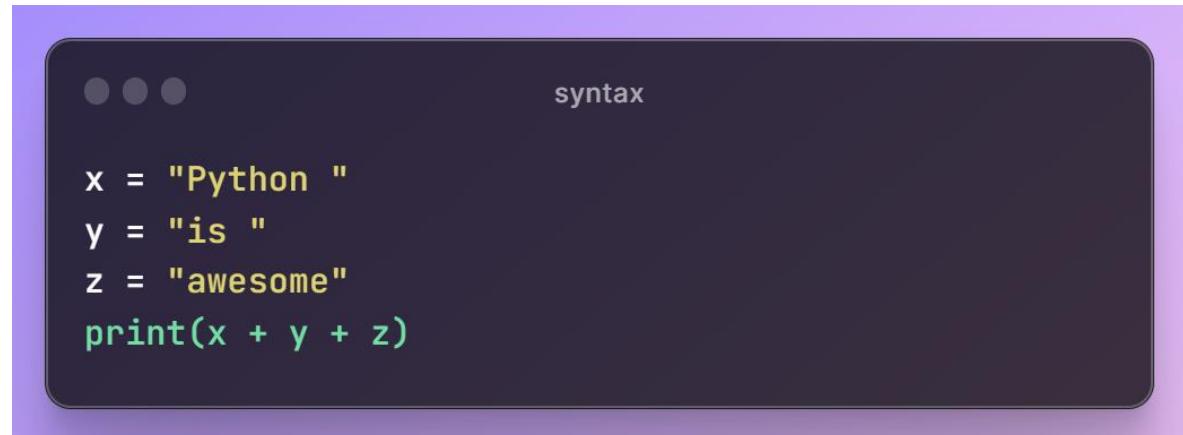


A screenshot of a code editor window with a dark theme. In the top right corner, the word "syntax" is displayed above three small circular icons. The main area contains the following Python code:
x = "Python"
y = "is"
z = "awesome"
print(x, y, z)

PYTHON

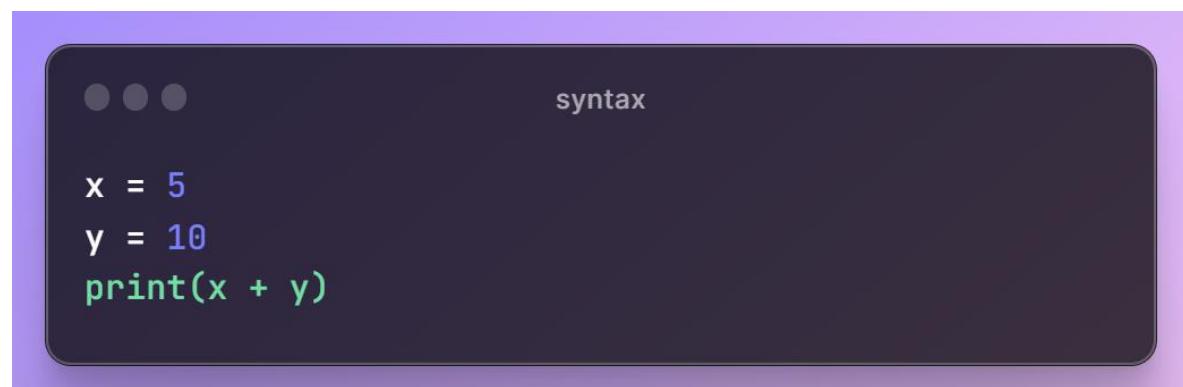
Output Variables

- You can also use the `+` operator to output multiple variables:
- Notice the space character after "Python " and "is ", without them the result would be "Pythonisawesome".
- For numbers, the `+` character works as a mathematical operator:



The screenshot shows a dark-themed code editor window. At the top left are three gray circular icons. To the right of them is the word "syntax". Below this, there is a block of Python code:

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



The screenshot shows a dark-themed code editor window. At the top left are three gray circular icons. To the right of them is the word "syntax". Below this, there is a block of Python code:

```
x = 5
y = 10
print(x + y)
```

PYTHON

Output Variables

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

```
... ● ● ● syntax  
x = 5  
y = "John"  
print(x + y)
```

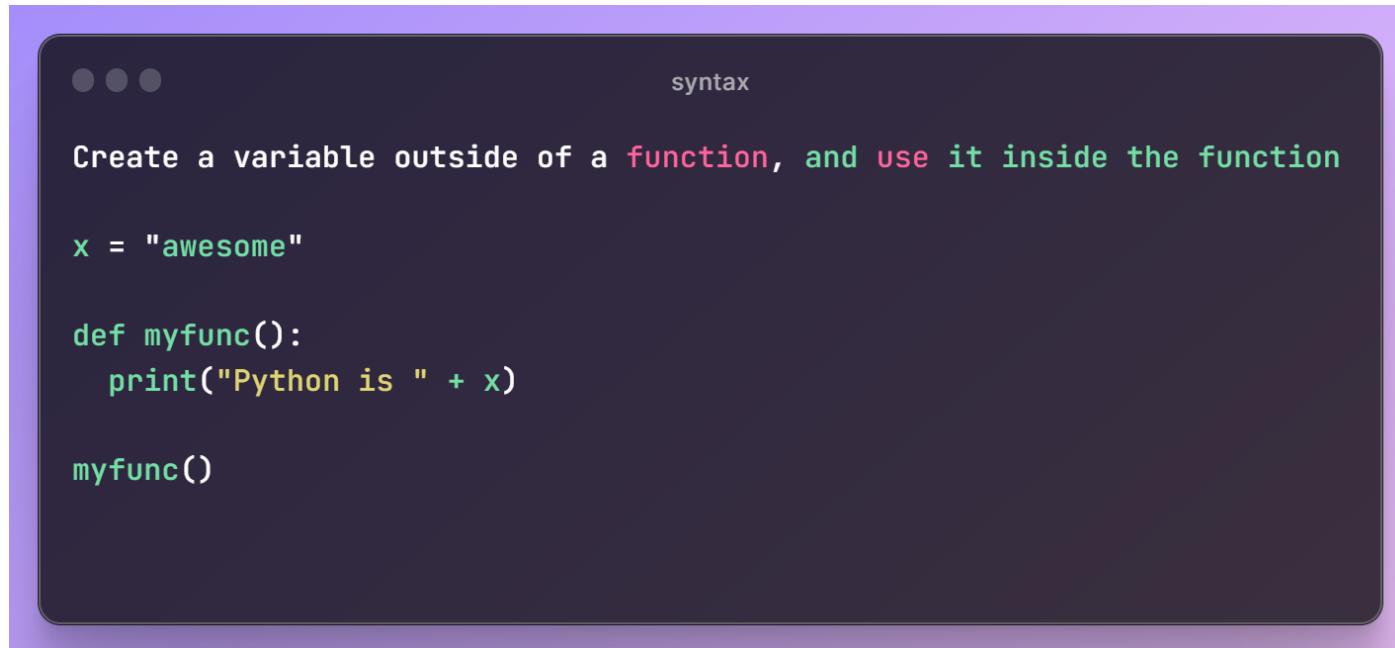
The best way to output multiple variables in the `print()` function is to separate them with commas, which even support different data types:

```
... ● ● ● syntax  
x = 5  
y = "John"  
print(x, y)
```

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.

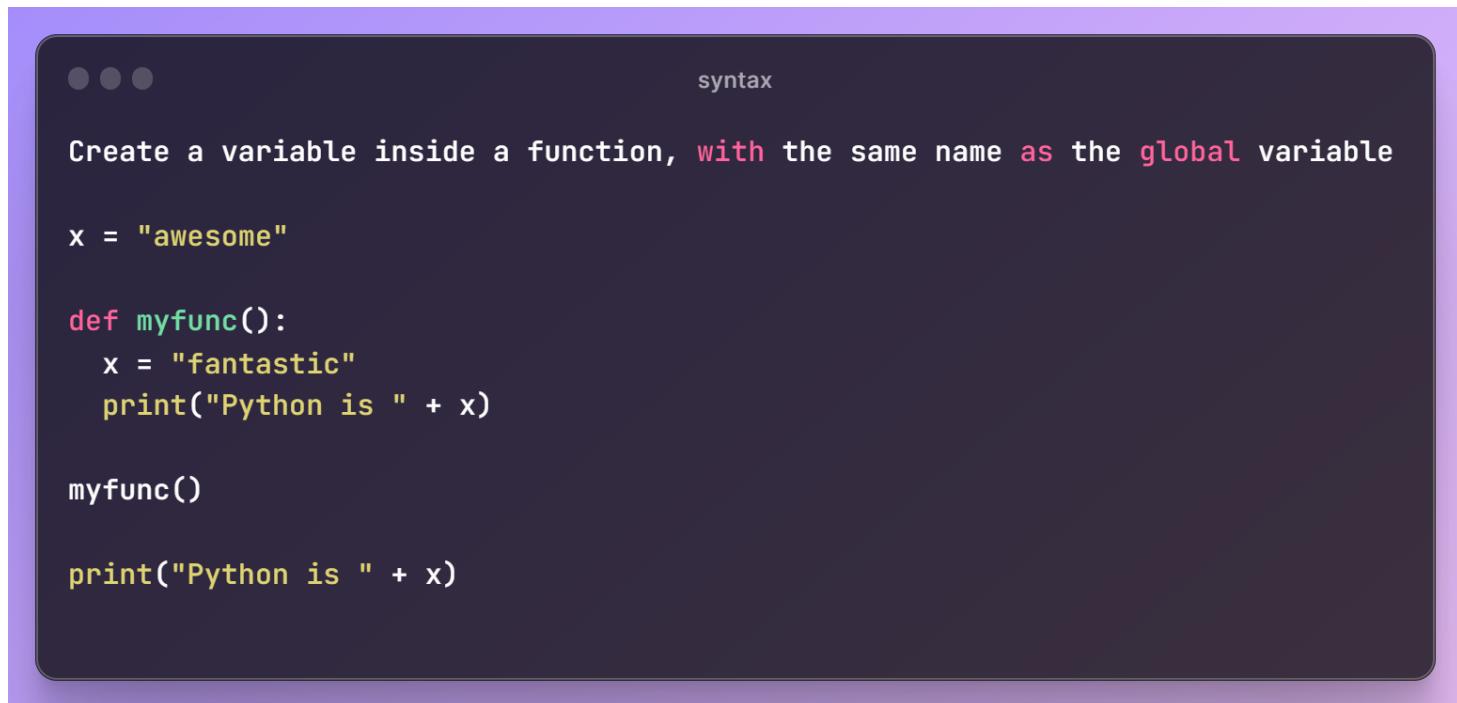


The image shows a screenshot of a Python code editor. At the top left, there are three small circular icons. To the right, the word "syntax" is displayed. The main area contains the following Python code:

```
...  
Create a variable outside of a function, and use it inside the function  
  
x = "awesome"  
  
def myfunc():  
    print("Python is " + x)  
  
myfunc()
```

Global Variables

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.



```
... syntax

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

x = "awesome"

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

myfunc()

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

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.

```
... syntax  
If you use the global keyword, the variable belongs to the global scope:  
  
def myfunc():  
    global x  
    x = "fantastic"  
  
myfunc()  
  
print("Python is " + x)
```

```
... syntax  
To change the value of a global variable inside a function, refer to the variable by using the global keyword:  
  
x = "awesome"  
  
def myfunc():  
    global x  
    x = "fantastic"  
  
myfunc()  
  
print("Python is " + x)
```

- Also, use the **global** keyword if you want to change a global variable inside a function.

PYTHON

Data types

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: str

Numeric Types: int, float, complex

Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

Boolean Type: bool

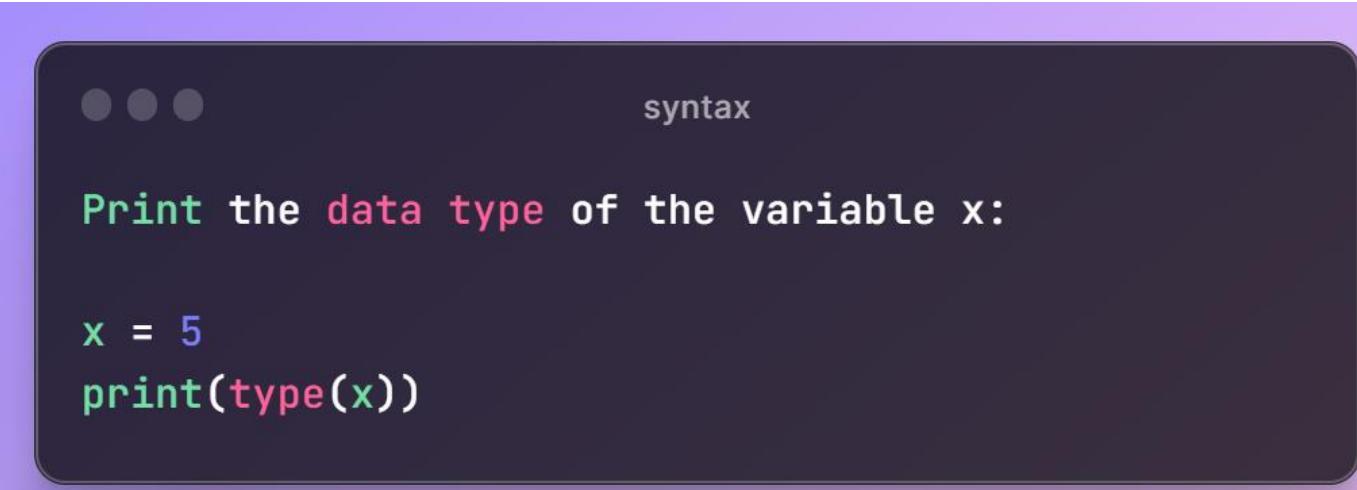
Binary Types: bytes, bytearray, memoryview

None Type: NoneType

PYTHON

Data types

You can get the data type of any object by using the `type()` function:



syntax

```
Print the data type of the variable x:  
  
x = 5  
print(type(x))
```

PYTHON

Data types

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset

PYTHON

Data types

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview
x = None	NoneType

PYTHON

Data types

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset

PYTHON

Data types

x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

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:

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

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

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

PYTHON

Int

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

```
... syntax

x = 1
y = 35656222554887711
z = -3255522

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

PYTHON

Float

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

Float can also be scientific numbers with an "e" to indicate the power of 10.



syntax

FLOATS:

```
x = 1.10  
y = 1.0  
z = -35.59
```

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



syntax

FLOATS:

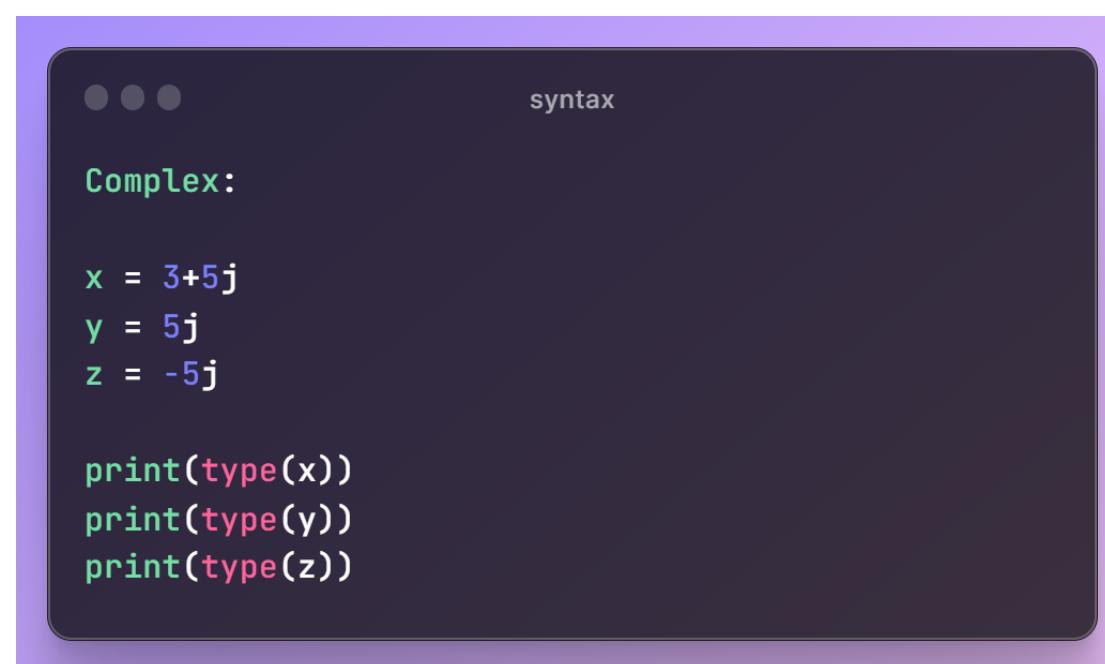
```
x = 35e3  
y = 12E4  
z = -87.7e100
```

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

PYTHON

Complex

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



The image shows a screenshot of a Python code editor. At the top right, there are three small circular icons. To their right, the word "syntax" is displayed. Below this, the text "Complex:" is shown in green. Underneath, there are three assignments: "x = 3+5j", "y = 5j", and "z = -5j". At the bottom, there are three print statements: "print(type(x))", "print(type(y))", and "print(type(z))". The entire code block is highlighted with a purple gradient background.

```
Complex:  
x = 3+5j  
y = 5j  
z = -5j  
  
print(type(x))  
print(type(y))  
print(type(z))
```

Type Conversion

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

Note: You cannot convert complex numbers into another number type.

```
... syntax

Convert from one type to another:

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

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

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

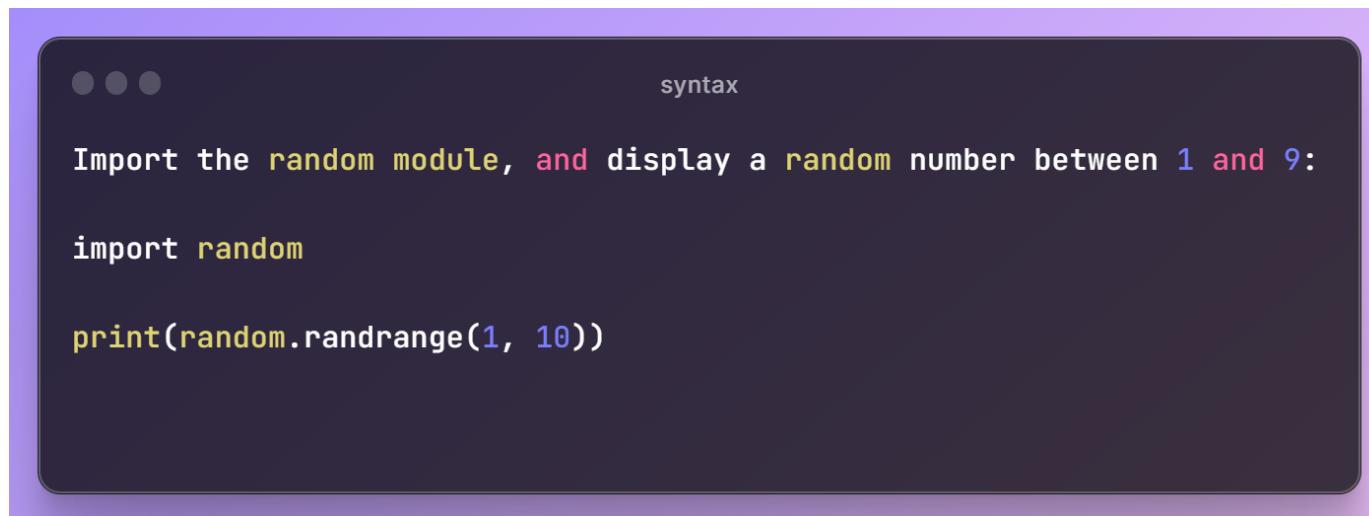
#convert from int to complex:
c = complex(x)

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

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

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:



The image shows a terminal window with a dark background and light-colored text. At the top left, there are three small circular icons. To the right of them, the word "syntax" is written in a small font. Below this, a descriptive message reads: "Import the random module, and display a random number between 1 and 9:". Underneath the message, two lines of Python code are shown: "import random" and "print(random.randrange(1, 10))".

```
... syntax

Import the random module, and display a random number between 1 and 9:

import random

print(random.randrange(1, 10))
```

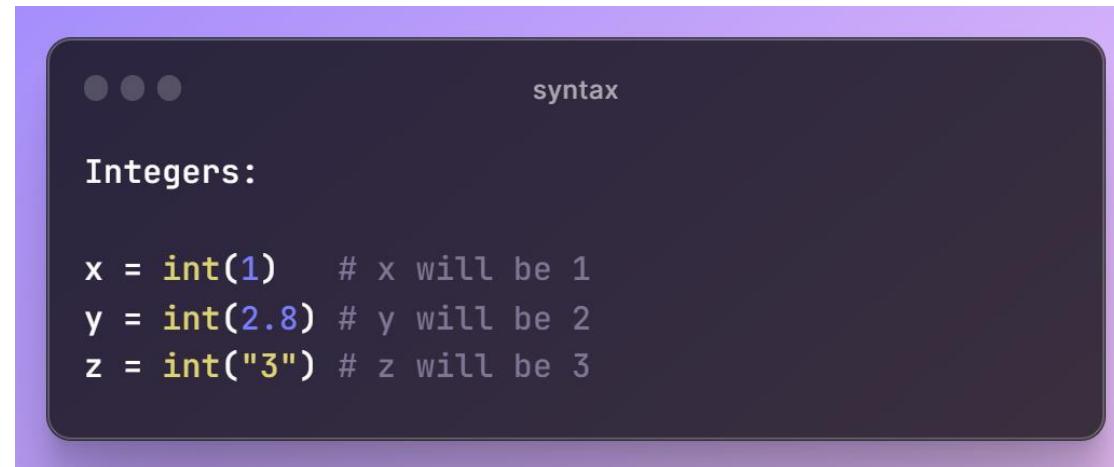
PYTHON

Casting

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)



The image shows a screenshot of a Python code editor. At the top right, there is a 'syntax' button. Below it, the word 'Integers:' is displayed. Underneath, three lines of Python code are shown, each demonstrating the use of the int() constructor:

```
x = int(1)    # x will be 1
y = int(2.8)  # y will be 2
z = int("3")  # z will be 3
```

PYTHON

Casting

- **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)

... syntax

Floats:

```
x = float(1)      # x will be 1.0
y = float(2.8)    # y will be 2.8
z = float("3")    # z will be 3.0
w = float("4.2")  # w will be 4.2
```

... syntax

Strings:

```
x = str("s1") # x will be 's1'
y = str(2)     # y will be '2'
z = str(3.0)   # z will be '3.0'
```

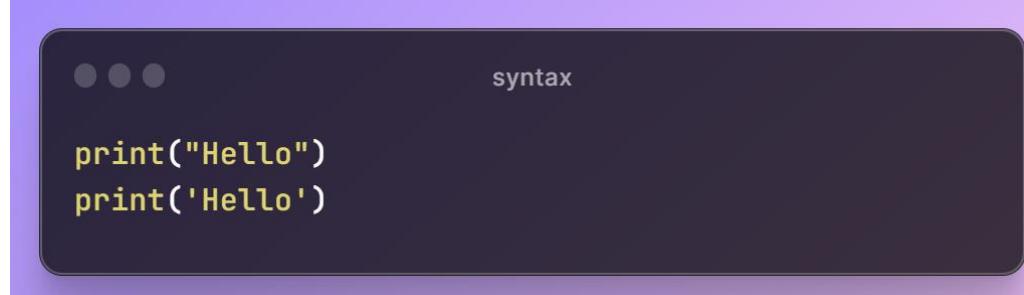
PYTHON

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

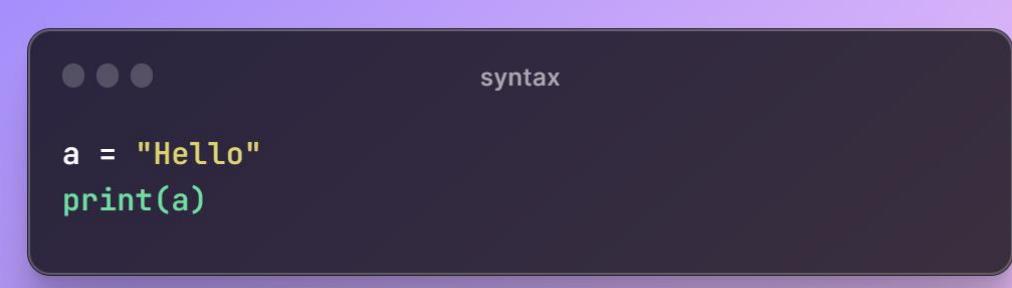
'hello' is the same as "hello".

You can display a string literal with the `print()` function:



```
... syntax  
print("Hello")  
print('Hello')
```

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:



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

Multiline Strings

You can assign a multiline string to a variable by using three quotes:



syntax

```
You can use three double quotes:  
  
a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""  
print(a)
```

Or three single quotes:



syntax

```
a = '''Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.'''  
print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

PYTHON

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.

The image shows a dark-themed code editor window with a purple header bar. In the top right corner of the header bar, there are three small circular icons. To the right of these icons, the word "syntax" is written in a light gray font. The main area of the editor contains the following text:

```
...  
Get the character at position 1 (remember that the first character has the position 0):  
  
a = "Hello, World!"  
print(a[1])
```

PYTHON

Loops Through a String

Since strings are arrays, we can loop through the characters in a string, with a **for** loop.

... syntax

Loop through the letters in the word "banana":

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

String Length

To get the length of a string, use the **len()** function.

... syntax

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

```
a = "Hello, World!"  
print(len(a))
```

PYTHON

Check String

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



The image shows a dark-themed code editor window. At the top left are three small circular icons. To the right of them, the word "syntax" is written in a light color. Below this, there is a snippet of Python code. It starts with three dots followed by the word "syntax". Then it says "Check if 'free' is present in the following text:". Below that, the code defines a variable "txt" with the value "The best things in life are free!" and then prints the result of the "free" keyword being used in an "in" statement.

```
... syntax  
Check if "free" is present in the following text:  
  
txt = "The best things in life are free!"  
print("free" in txt)
```

Use it in an **if** statement:



This image shows another dark-themed code editor window with three circular icons at the top left and the word "syntax" to its right. It contains a snippet of Python code. It begins with three dots, followed by "syntax", then "Print only if 'free' is present:", and finally a block of code that defines "txt" as the previous string, uses an "if" statement to check if "free" is in "txt", and prints a message if it is found.

```
... syntax  
Print only if "free" is present:  
  
txt = "The best things in life are free!"  
if "free" in txt:  
    print("Yes, 'free' is present.")
```

PYTHON

Check If Not

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

Check **if "expensive" is NOT present in** the following text:

txt = "The best things in life are free!"
print("expensive" **not in** txt)

Use it in an **if** statement:

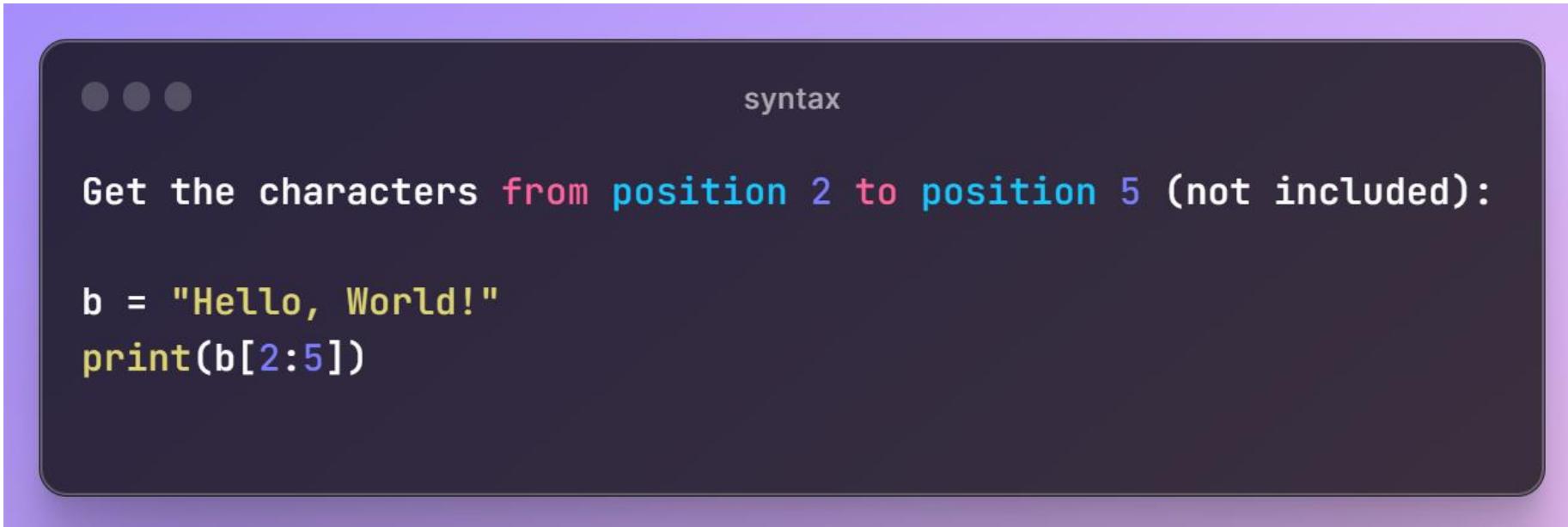
print **only if "expensive" is NOT present:**

txt = "The best things in life are free!"
if "expensive" not in txt:
 print("No, 'expensive' is NOT present.")

Slicing Strings

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.



The image shows a dark-themed code editor window with a purple header bar. The header bar has three dots on the left and the word "syntax" on the right. The main area of the window contains the following text:

```
Get the characters from position 2 to position 5 (not included):  
  
b = "Hello, World!"  
print(b[2:5])
```

Note: The first character has index 0.

PYTHON

Slicing Strings

Slice From the Start

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



syntax

```
Get the characters from the start to position 5 (not included):  
  
b = "Hello, World!"  
print(b[:5])
```

Slice To the End

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

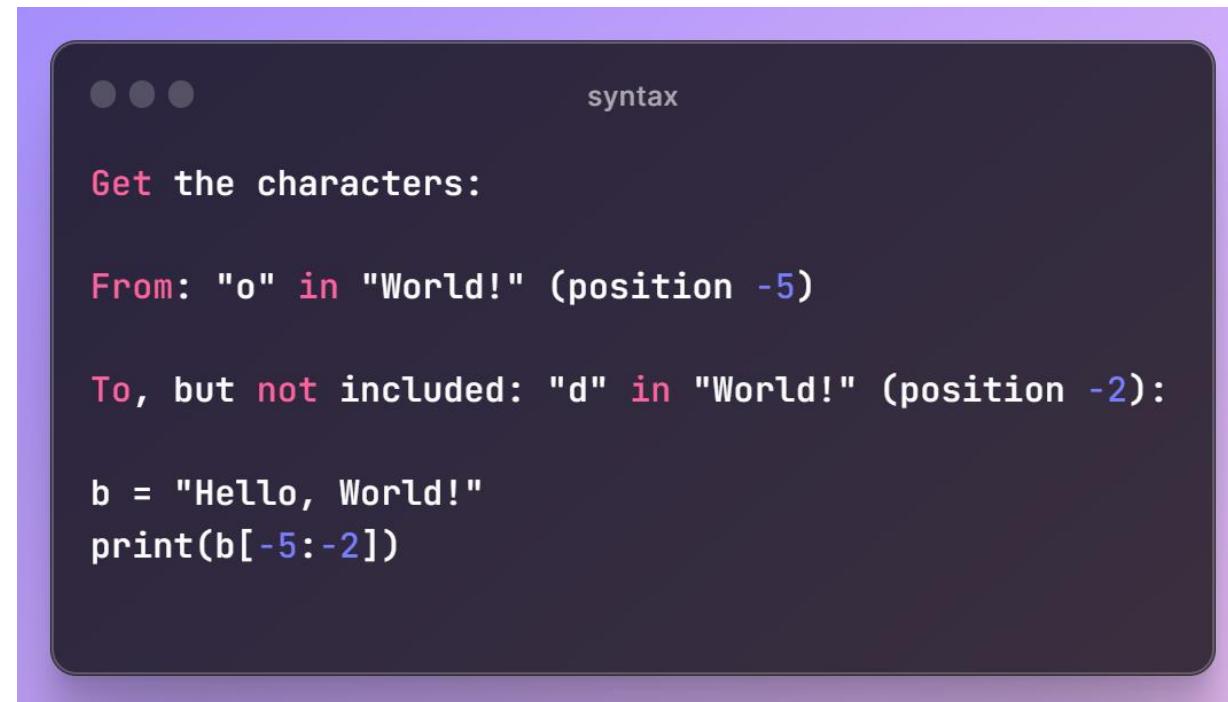


syntax

```
Get the characters from position 2, and all the way to the end:  
  
b = "Hello, World!"  
print(b[2:])
```

Negative Strings

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



Modify Strings

Upper Case



syntax

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

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

Lower Case



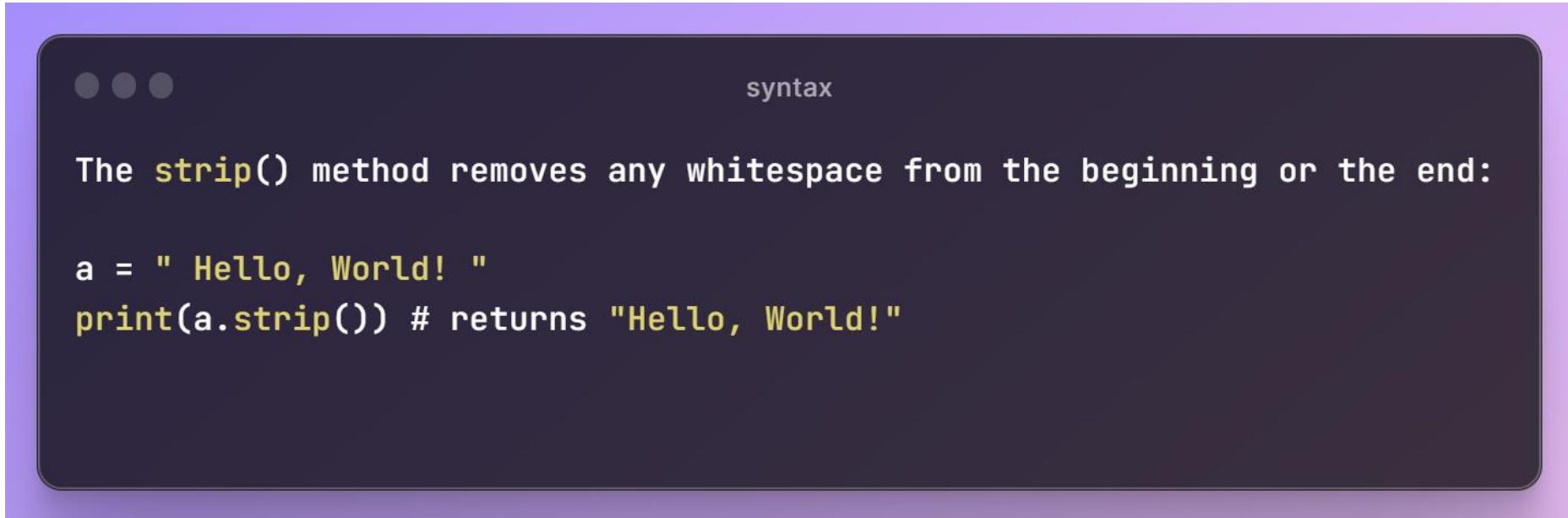
syntax

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

```
a = "Hello, World!"  
print(a.lower())
```

Remove Whitespace

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



The screenshot shows a terminal window with a dark background and light-colored text. At the top left, there are three small circular icons. To the right of them, the word "syntax" is written in a smaller font. Below this, a explanatory text block starts with "The **strip()** method removes any whitespace from the beginning or the end:". Underneath, there is a code example:

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

Modify Strings

Replace String

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

Split string

... syntax
The **replace()** method replaces a string with another string:

```
a = "Hello, World!"  
print(a.replace("H", "J"))
```

... syntax
The **split()** method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"  
print(a.split(",")) # returns ['Hello', ' World!']
```

PYTHON

String Concatenation

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



syntax

Merge variable **a** with variable **b** into variable **c**:

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



syntax

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

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

PYTHON

String Format

- As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

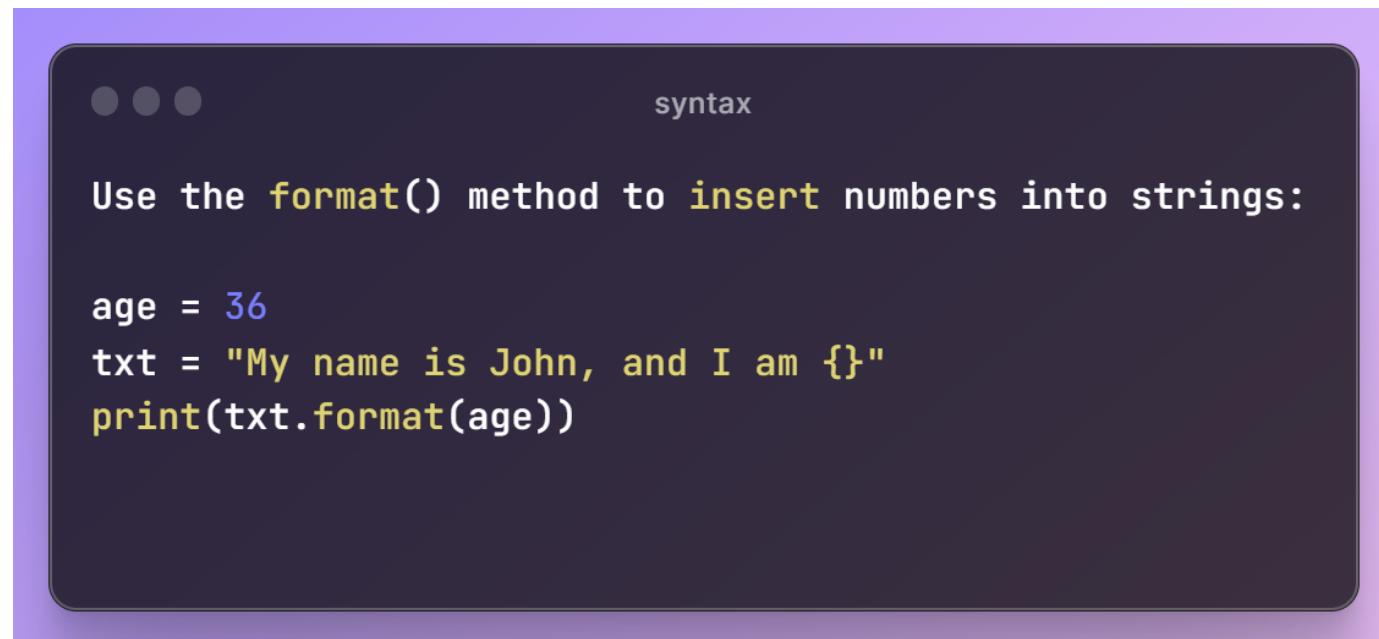
The image shows a screenshot of a Python code editor. The code is as follows:

```
...syntax  
age = 36  
txt = "My name is John, I am " + age  
print(txt)
```

The code editor has a dark theme with light-colored text. A tooltip or callout box is overlaid on the screen, containing three dots at the top left and the word "syntax" at the top right, pointing towards the first line of code.

String Format

- But we can combine strings and numbers by using the **format()** method!
- The **format()** method takes the passed arguments, formats them, and places them in the string where the placeholders **{}** are:



String Format

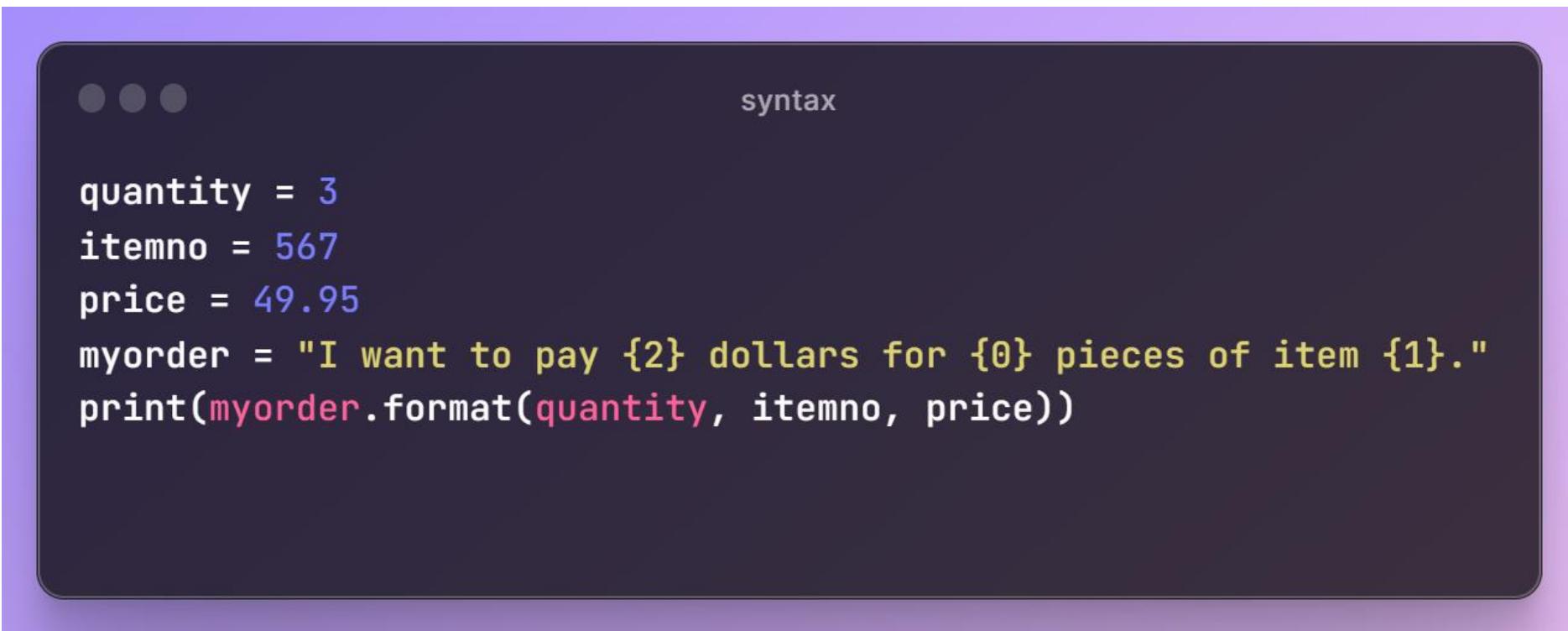
- The `format()` method takes unlimited number of arguments, and are placed into the respective placeholders:

```
...                                     syntax

quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

String Format

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



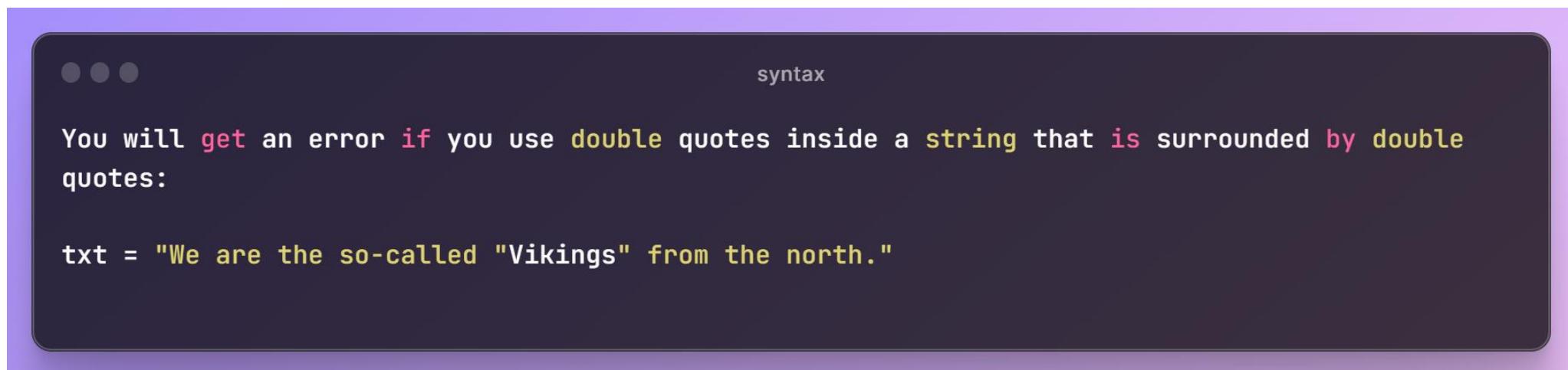
The image shows a screenshot of a Python code editor. The code is enclosed in a dark gray rounded rectangle with a thin white border. In the top right corner of this box, the word "syntax" is written in a small, light gray font. In the top left corner, there are three small gray circular dots. The code itself is written in a light gray font on a dark background. It defines variables for quantity, itemno, and price, and then creates a string myorder using the format method to insert these variables at specific indices {0}, {1}, and {2}.

```
... syntax

quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Escape Characters

- 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:

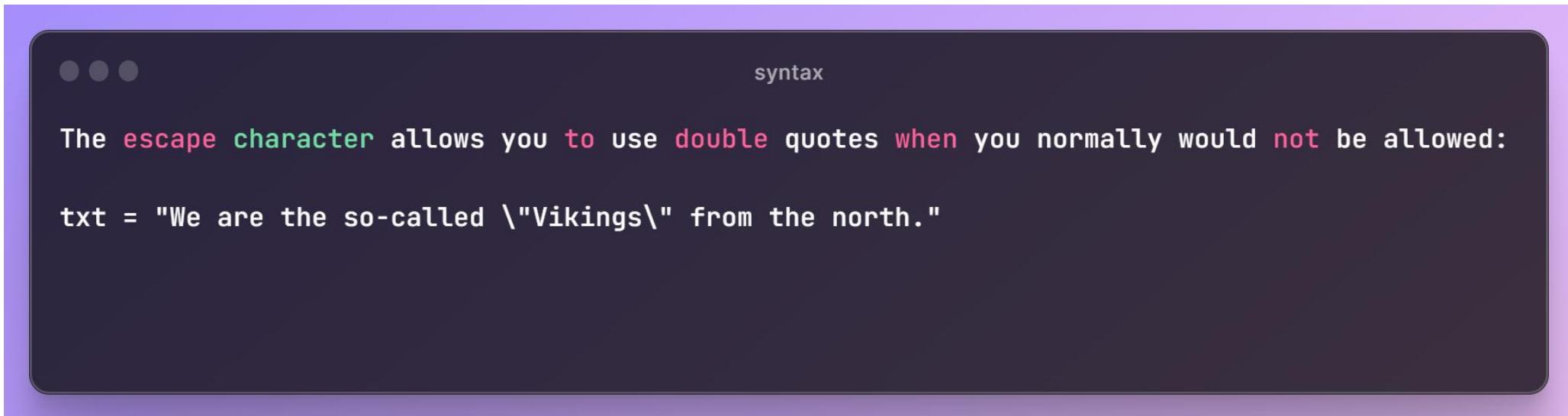


The screenshot shows a terminal window with a dark background and light-colored text. At the top left, there are three small circular icons. On the right, the word "syntax" is written in a smaller font. Below this, a message is displayed: "You will get an error if you use double quotes inside a string that is surrounded by double quotes:". At the bottom, a line of Python code is shown: "txt = "We are the so-called "Vikings" from the north."". The code is highlighted in yellow, while the error message is in white.

PYTHON

Escape Characters

- To fix this problem, use the escape character \":



PYTHON

Escape Characters

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

PYTHON

String Methods

- Python has a set of built-in methods that you can use on strings.
- **Note:** All string methods return new values. They do not change the original string.

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

PYTHON

String Methods

<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>isascii()</u>	Returns True if all characters in the string are ascii characters
<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

PYTHON

String Methods

<u>ljust()</u>	Returns a left justified version of the string
<u>lower()</u>	Converts a string into lower case
<u>lstrip()</u>	Returns a left trim version of the string
<u>maketrans()</u>	Returns a translation table to be used in translations
<u>partition()</u>	Returns a tuple where the string is parted into three parts
<u>replace()</u>	Returns a string where a specified value is replaced with a specified value
<u>rfind()</u>	Searches the string for a specified value and returns the last position of where it was found
<u>rindex()</u>	Searches the string for a specified value and returns the last position of where it was found
<u>rjust()</u>	Returns a right justified version of the string
<u>rpartition()</u>	Returns a tuple where the string is parted into three parts

PYTHON

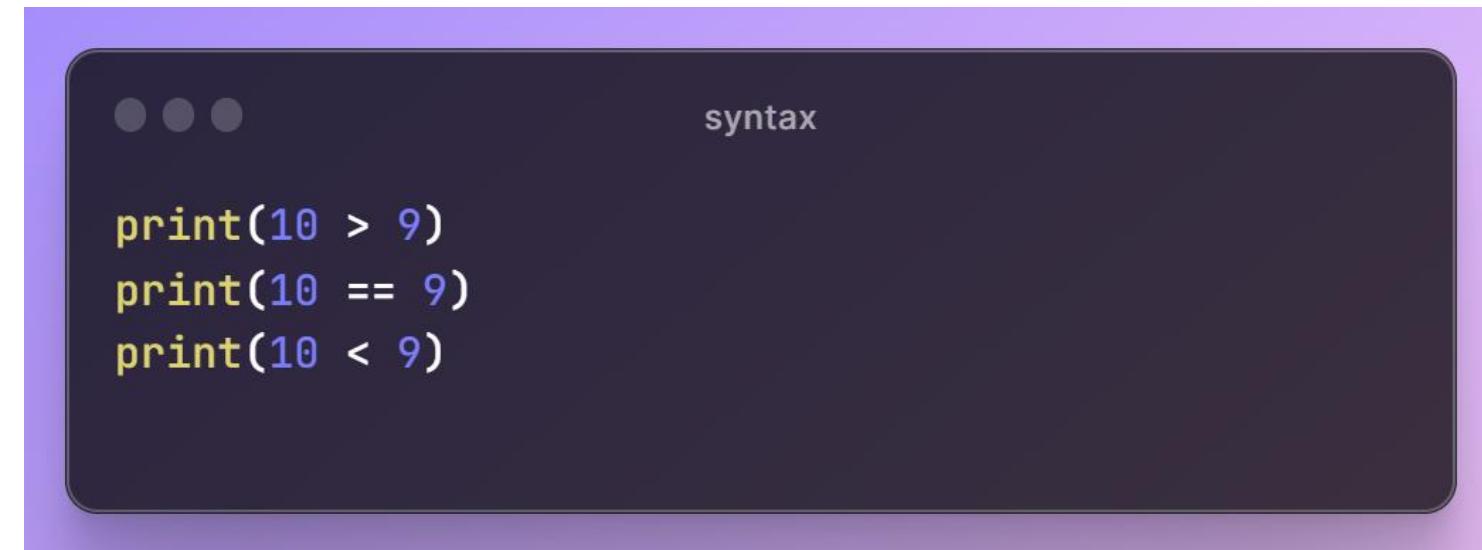
String Methods

<u>rsplit()</u>	Splits the string at the specified separator, and returns a list
<u>rstrip()</u>	Returns a right trim version of the string
<u>split()</u>	Splits the string at the specified separator, and returns a list
<u>splitlines()</u>	Splits the string at line breaks and returns a list
<u>startswith()</u>	Returns true if the string starts with the specified value
<u>strip()</u>	Returns a trimmed version of the string
<u>swapcase()</u>	Swaps cases, lower case becomes upper case and vice versa
<u>title()</u>	Converts the first character of each word to upper case
<u>translate()</u>	Returns a translated string
<u>upper()</u>	Converts a string into upper case
<u>zfill()</u>	Fills the string with a specified number of 0 values at the beginning

PYTHON

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:



The image shows a dark-themed code editor window with a purple header bar. The header bar has three dots on the left and the word "syntax" on the right. The main area of the editor contains the following Python code:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

PYTHON

Boolean Values

- When you run a condition in an if statement, Python returns **True** or **False**:

The image shows a screenshot of a Python code editor. The code is enclosed in a dark purple rounded rectangle with a light purple border. At the top left, there are three small circular icons. To the right of them, the word "syntax" is written in a small, light-colored font. Below this, the code itself is displayed in white text on a dark background. The code prints a message based on a comparison between two variables, 'a' and 'b'. If 'b' is greater than 'a', it prints "b is greater than a"; otherwise, it prints "b is not greater than a".

```
... syntax

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

a = 200
b = 33

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

PYTHON

Evaluate Values and Variables

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



syntax

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
```



syntax

Evaluate two variables:

```
x = "Hello"
y = 15

print(bool(x))
print(bool(y))
```

PYTHON

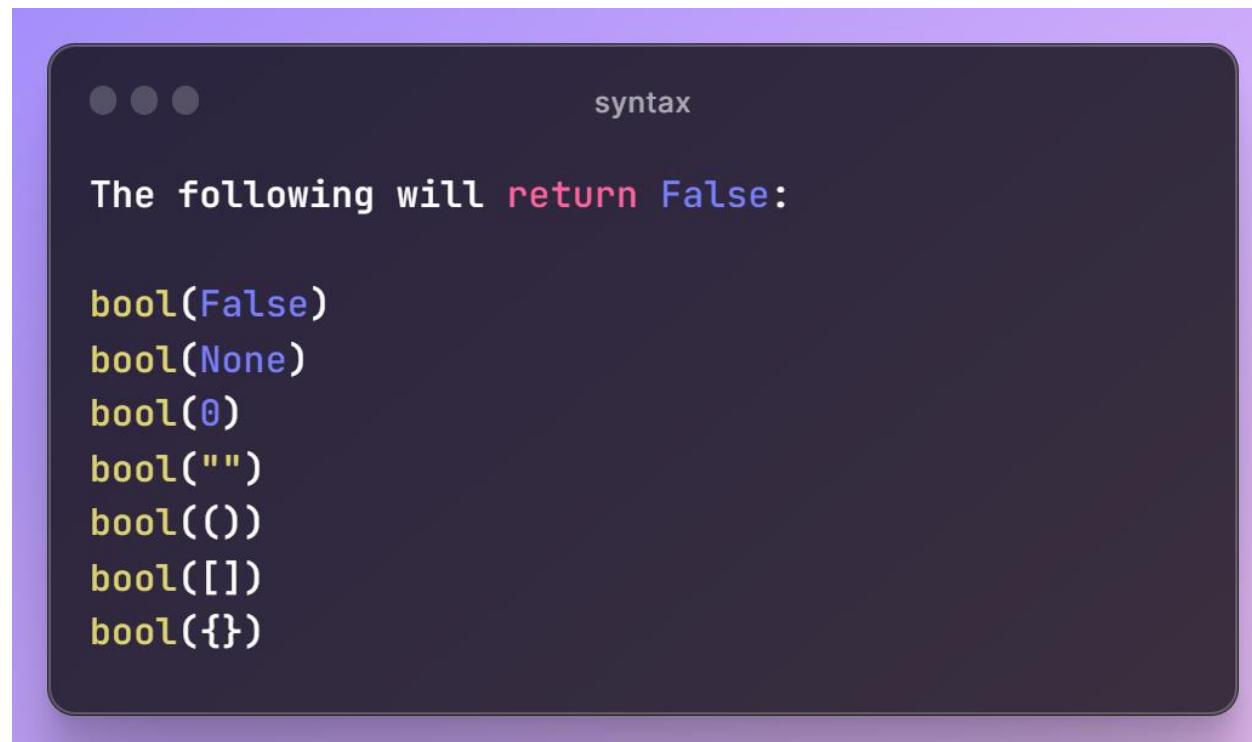
Most Values are true

- 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.



Some Values are false

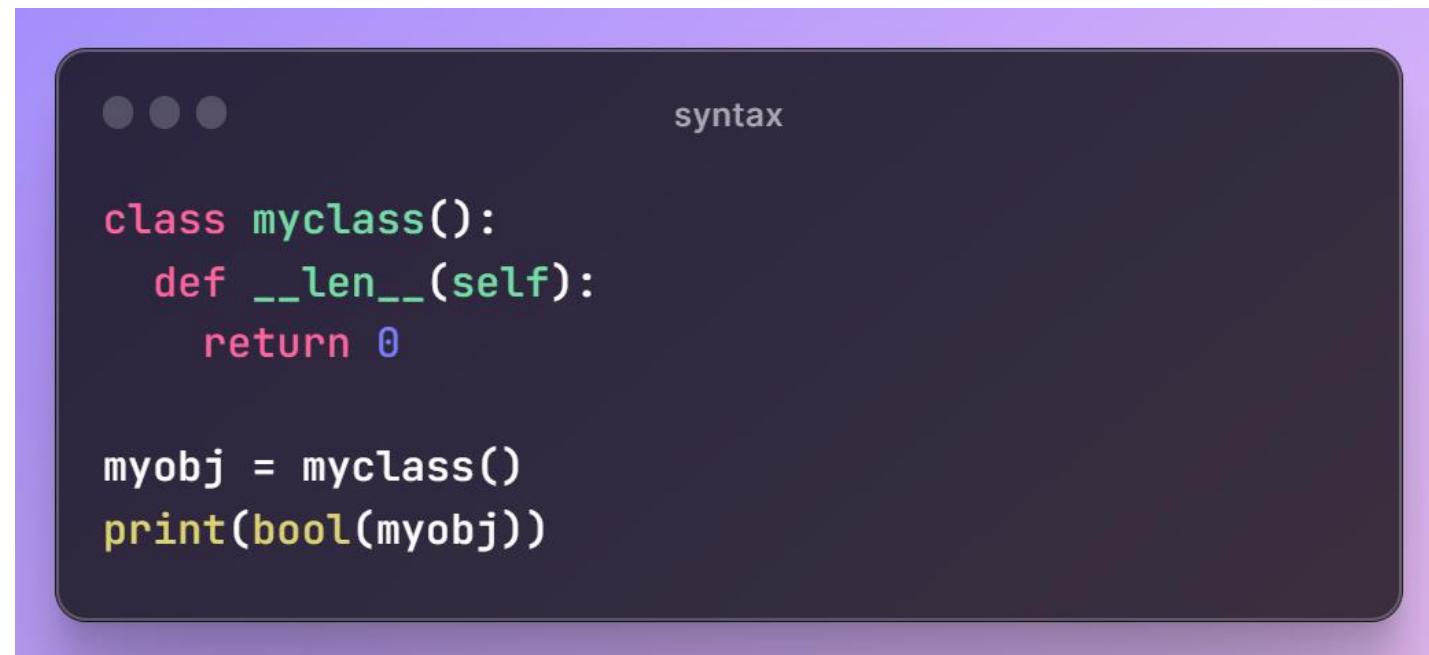
- In fact, there are not many values that evaluate to **False**, except empty values, such as `()`, `[]`, `{}`, `""`, the number `0`, and the value `None`. And of course the value **False** evaluates to **False**.



PYTHON

Some Values are 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**:



```
... syntax

class MyClass():
    def __len__(self):
        return 0

myobj = MyClass()
print(bool(myobj))
```

PYTHON

Function can Return a Boolean

- You can create functions that returns a Boolean Value:
- You can execute code based on the Boolean answer of a function:

```
... syntax  
Print the answer of a function:  
  
def myFunction() :  
    return True  
  
print(myFunction())
```

```
... syntax  
Print "YES!" if the function returns True, otherwise print "NO!":  
  
def myFunction() :  
    return True  
  
if myFunction():  
    print("YES!")  
else:  
    print("NO!")
```

PYTHON

Function can Return a Boolean

- 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:



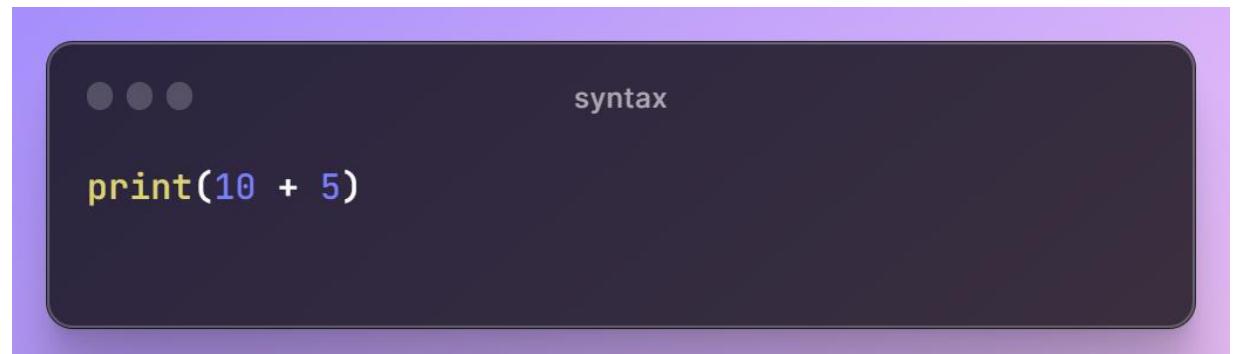
PYTHON

Operators

- Operators are used to perform operations on variables and values.
- In the example below, we use the `+` operator to add together two values:

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

PYTHON

Assignment Operators

Assignment operators are used to assign values to variables:

<code>**=</code>	<code>x **= 3</code>	<code>x = x ** 3</code>
<code>&=</code>	<code>x &= 3</code>	<code>x = x & 3</code>
<code> =</code>	<code>x = 3</code>	<code>x = x 3</code>
<code>^=</code>	<code>x ^= 3</code>	<code>x = x ^ 3</code>
<code>>>=</code>	<code>x >>= 3</code>	<code>x = x >> 3</code>
<code><<=</code>	<code>x <<= 3</code>	<code>x = x << 3</code>

PYTHON

Comparison Operators

Comparison operators are used to compare two values:

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

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	x is y
is not	Returns True if both variables are not the same object	x is not y

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	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
	OR	Sets each bit to 1 if one of two bits is 1	x y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

PYTHON

Operator precedence

- Operator precedence describes the order in which operations are performed.



syntax

Parentheses has the highest precedence, meaning that expressions inside parentheses must be evaluated first:

```
print((6 + 3) - (6 + 3))
```



syntax

Multiplication * has higher precedence than addition +, and therefore multiplications are evaluated before additions:

```
print(100 + 5 * 3)
```

PYTHON

Operator precedence

- The precedence order is described in the table below, starting with the highest precedence at the top:

Operator	Description
<code>()</code>	Parentheses
<code>**</code>	Exponentiation
<code>+x</code> <code>-x</code> <code>~x</code>	Unary plus, unary minus, and bitwise NOT
<code>*</code> <code>/</code> <code>//</code> <code>%</code>	Multiplication, division, floor division, and modulus
<code>+</code> <code>-</code>	Addition and subtraction
<code><<</code> <code>>></code>	Bitwise left and right shifts
<code>&</code>	Bitwise AND

PYTHON

Operator precedence

- The precedence order is described in the table below, starting with the highest precedence at the top:

<code>^</code>	Bitwise XOR
<code> </code>	Bitwise OR
<code>== != > >= < <=</code> <code>is is not in not</code> <code>in</code>	Comparisons, identity, and membership operators
<code>not</code>	Logical NOT
<code>and</code>	AND
<code>or</code>	OR

Operator precedence

- If two operators have the same precedence, the expression is evaluated from left to right.



syntax

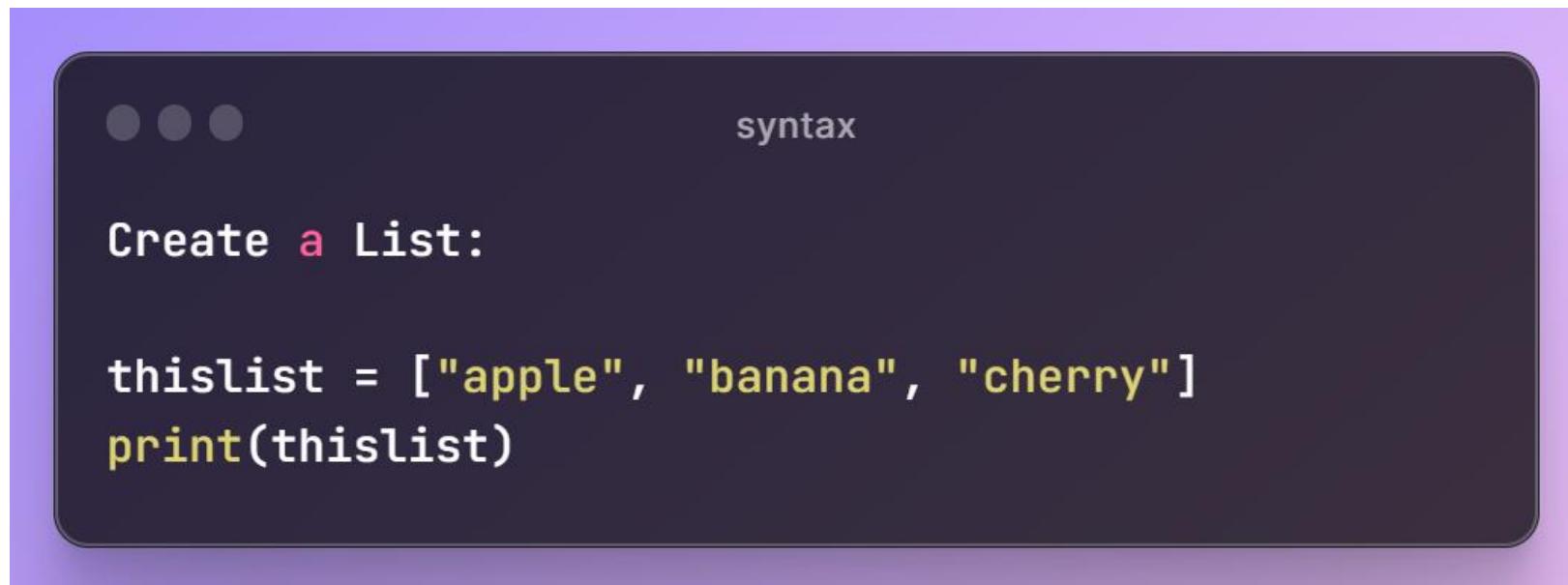
Addition + and subtraction - has the same precedence, and therefore we evaluate the expression from left to right:

```
print(5 + 4 - 7 + 3)
```

PYTHON

Lists

- 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: `mylist = ["apple", "banana", "cherry"]`



PYTHON

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.
- **Note:** There are some [list methods](#) that will change the order, but in general: the order of the items will not change.

Changeable

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

PYTHON

Allow Duplicates

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

```
... syntax  
Lists allow duplicate values:  
  
thislist = ["apple", "banana", "cherry", "apple", "cherry"]  
print(thislist)
```

List Length

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

```
... syntax  
Print the number of items in the list:  
  
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

List items – Data Types

- List items can be of any data type:

... syntax

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

... syntax

A list with strings, integers and boolean values:

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

PYTHON

type()

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

<class 'list'>



The image shows a dark-themed terminal window with a light purple header bar. In the top right corner of the header bar, there are three small white dots. To the right of these dots, the word "syntax" is written in a smaller white font. The main body of the terminal is dark gray. At the top left of this body, there are three small white dots. Below these dots, the text "What is the data type of a list?" is displayed in a white monospaced font. At the bottom of the terminal window, there is a block of Python code in a white monospaced font:
`myList = ["apple", "banana", "cherry"]
print(type(myList))`

The `list()` Constructor

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



syntax

Using the `list()` constructor to make a List:

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

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members.

*Set *items* are unchangeable, but you can remove and/or add items whenever you like.

**As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

PYTHON

Access List Item

- List items are indexed and you can access them by referring to the index number:



Note: The first item has index 0.

Negative Indexing

- Negative indexing means start from the end.
- -1 refers to the last item, -2 refers to the second last item etc.

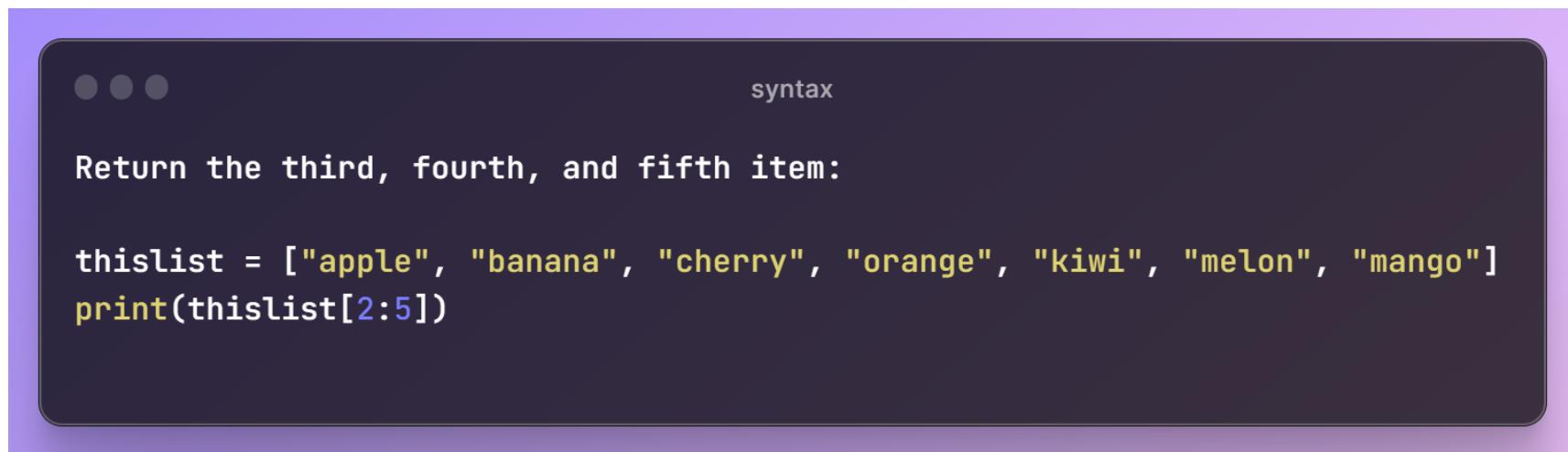


A terminal window with a dark background and light-colored text. It shows three dots at the top left, followed by the word "syntax". Below that, the text "Print the last item of the list:" is displayed. At the bottom, there is a code snippet:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Range Of Indexing

- 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.



```
... syntax  
Return the third, fourth, and fifth item:  
  
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:5])
```

Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0.

Range Of Indexing

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

```
... syntax  
This example returns the items from the beginning to, but NOT including, "kiwi":  
  
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[:4])
```

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

```
... syntax  
This example returns the items from "cherry" to the end:  
  
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:])
```

Range Of Negative Indexes

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



syntax

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

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

PYTHON

Check if Item Exists

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

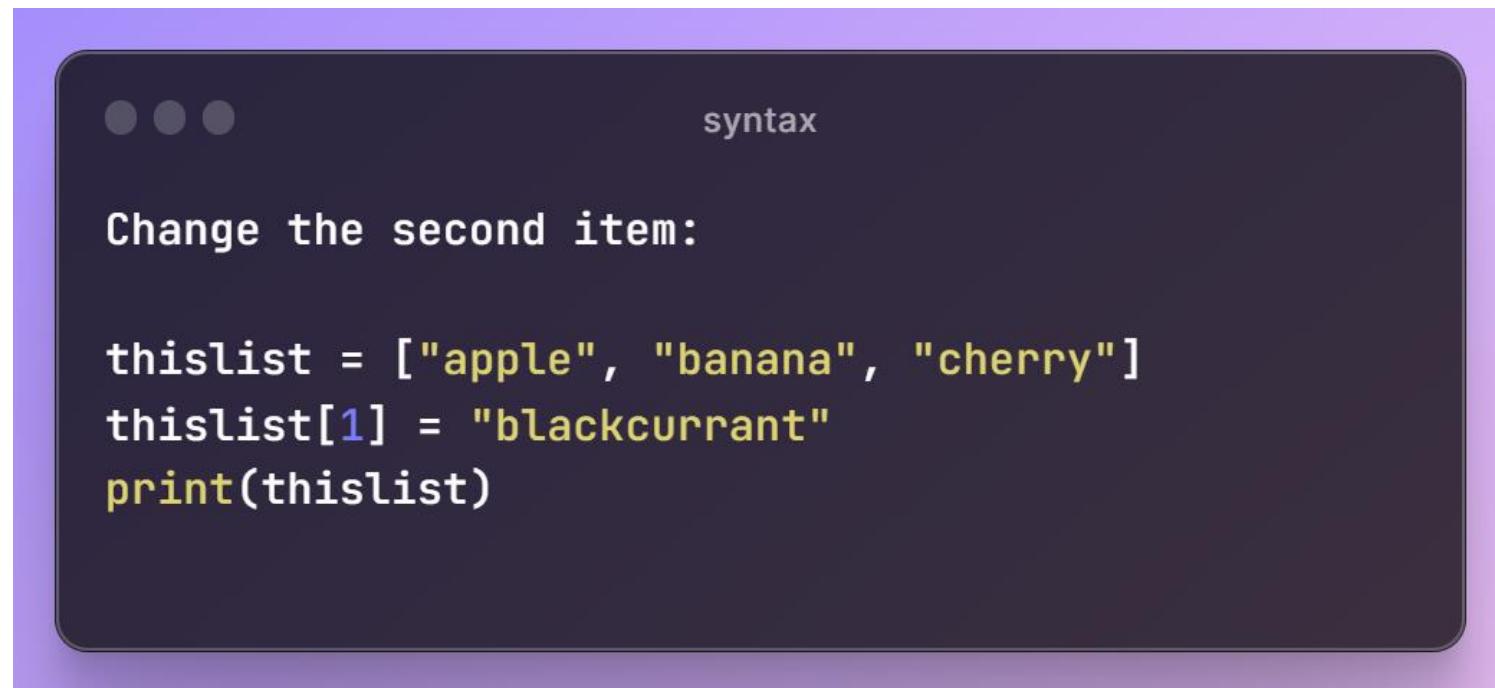


syntax

```
Check if "apple" is present in the list:  
  
thislist = ["apple", "banana", "cherry"]  
if "apple" in thislist:  
    print("Yes, 'apple' is in the fruits list")
```

Change Item Values

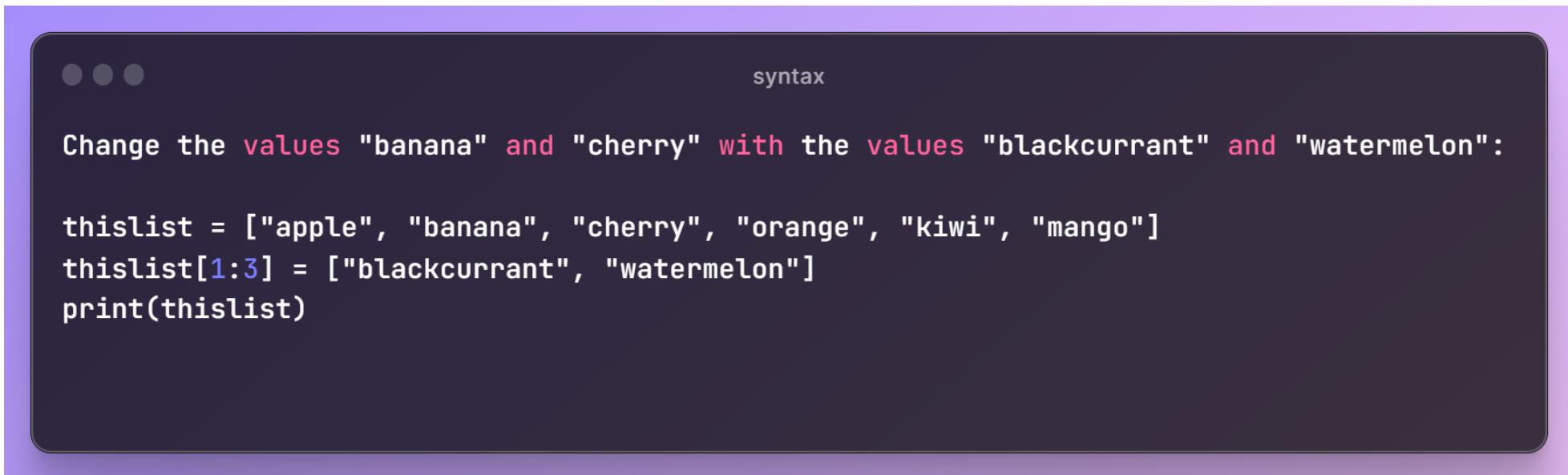
- To change the value of a specific item, refer to the index number:



PYTHON

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:



Change the `values` "banana" and "cherry" with the `values` "blackcurrant" and "watermelon":

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

Change a Range of Item Values

- If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:



syntax

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

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

Change a Range of Item Values

- **Note:** The length of the list will change when the number of items inserted does not match the number of items replaced.
- If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:



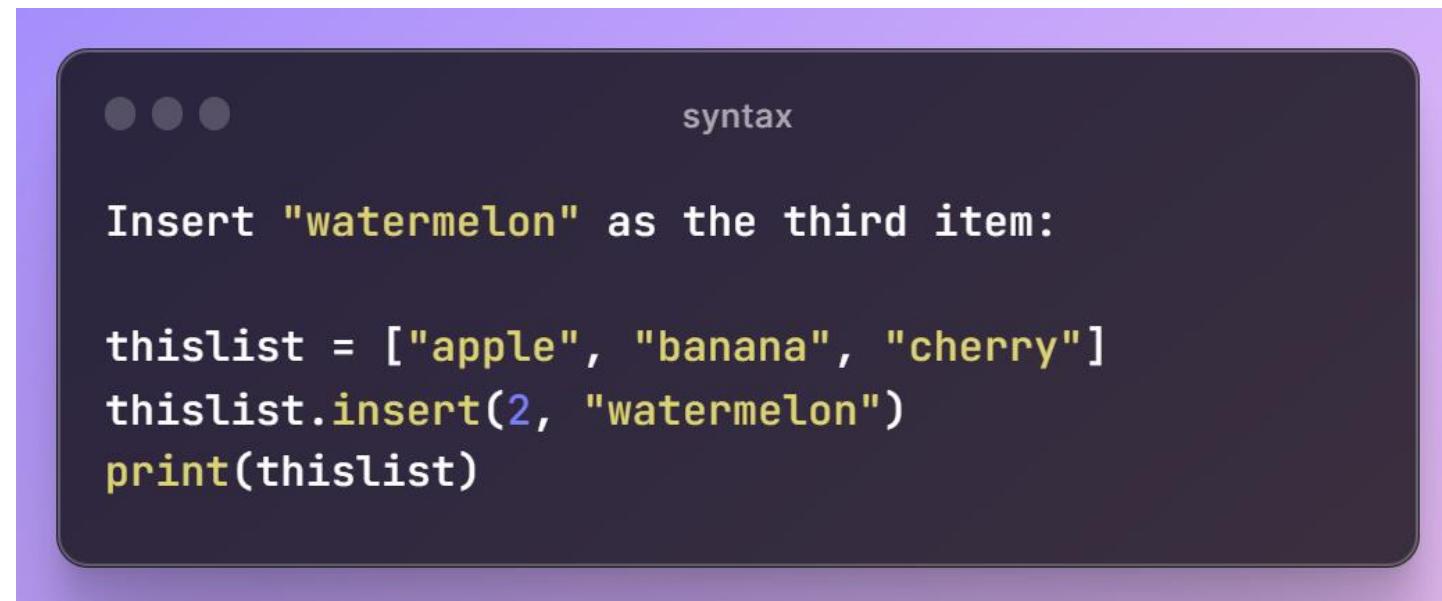
... syntax

Change the `second` and `third` value by replacing it with one value:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

Insert Item

- 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:



Note: As a result of the example above, the list will now contain 4 items.

Append Item

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



syntax

Using the **append()** method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Insert Item

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

Note: As a result of the examples above, the lists will now contain 4 items.



syntax

Insert an item as the second `position`:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Extend List

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



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.).

The screenshot shows a dark-themed code editor window. At the top left, there are three small circular icons. To the right of them, the word "syntax" is written in a light gray font. The main area of the editor contains the following code:

```
... syntax

Add elements of a tuple to a list:

thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

Remove Specified Item

- The **remove()** method removes the specified item.
- If there are more than one item with the specified value, the **remove()** method removes the first occurrence:

...

syntax

Remove "banana":

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

...

syntax

Remove the first occurrence of "banana":

```
thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)
```

Remove Specified Index

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



syntax

Remove the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```



syntax

Remove the last item:

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Remove Specified Index

- The **del** keyword also removes the specified index:
- The **del** keyword can also delete the list completely.



syntax

Remove the first item:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```



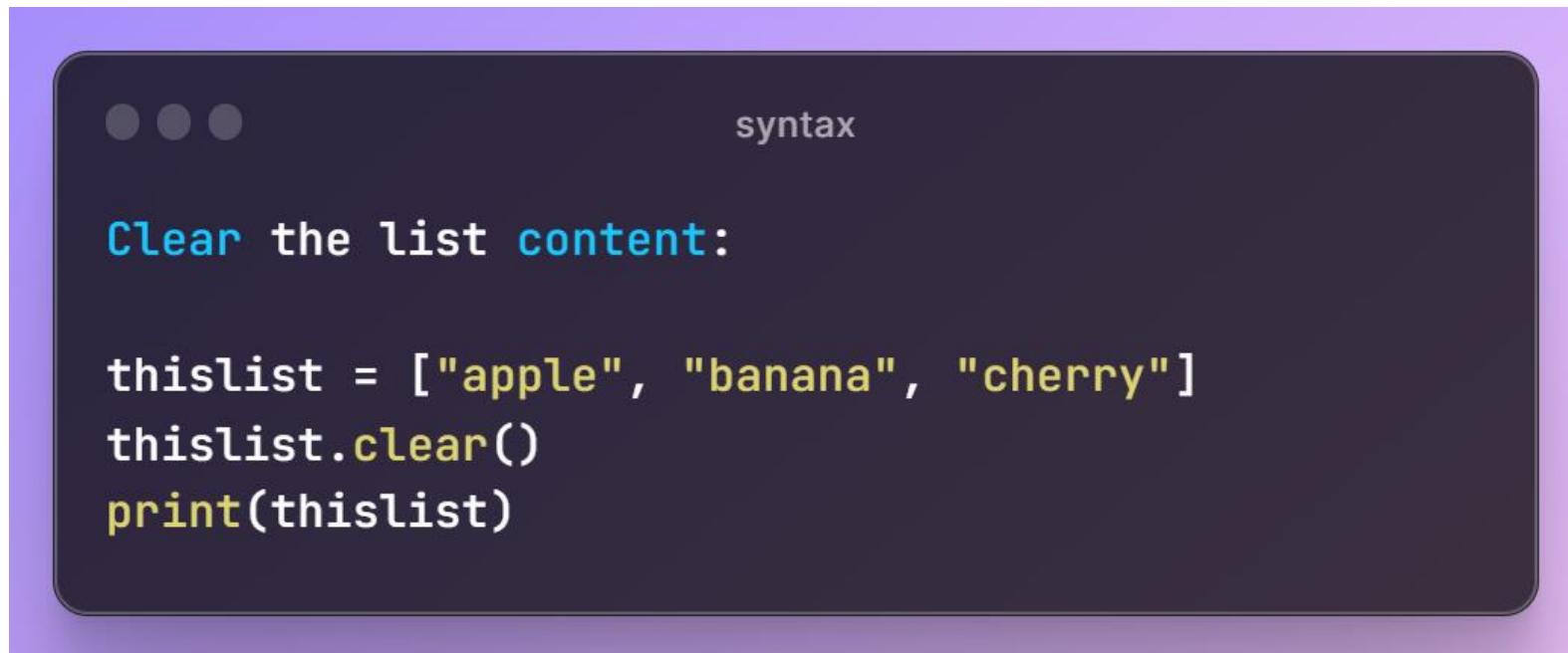
syntax

Delete the entire list:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Clear the List

- The `clear()` method empties the list.
- The list still remains, but it has no content.



The image shows a screenshot of a Python code editor. At the top left, there are three grey circular icons. To their right, the word "syntax" is written in a light blue font. Below this, the text "Clear the list content:" is displayed in a light blue font. Underneath, a list of code snippets is shown in white text on a dark background. The code consists of three lines:
`thislist = ["apple", "banana", "cherry"]`
`thislist.clear()`
`print(thislist)`

PYTHON

Loop through a List

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

syntax

Print all items in the list, one by one:

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

Loop through the index number

- 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.
- The iterable created in the example above is `[0, 1, 2]`.



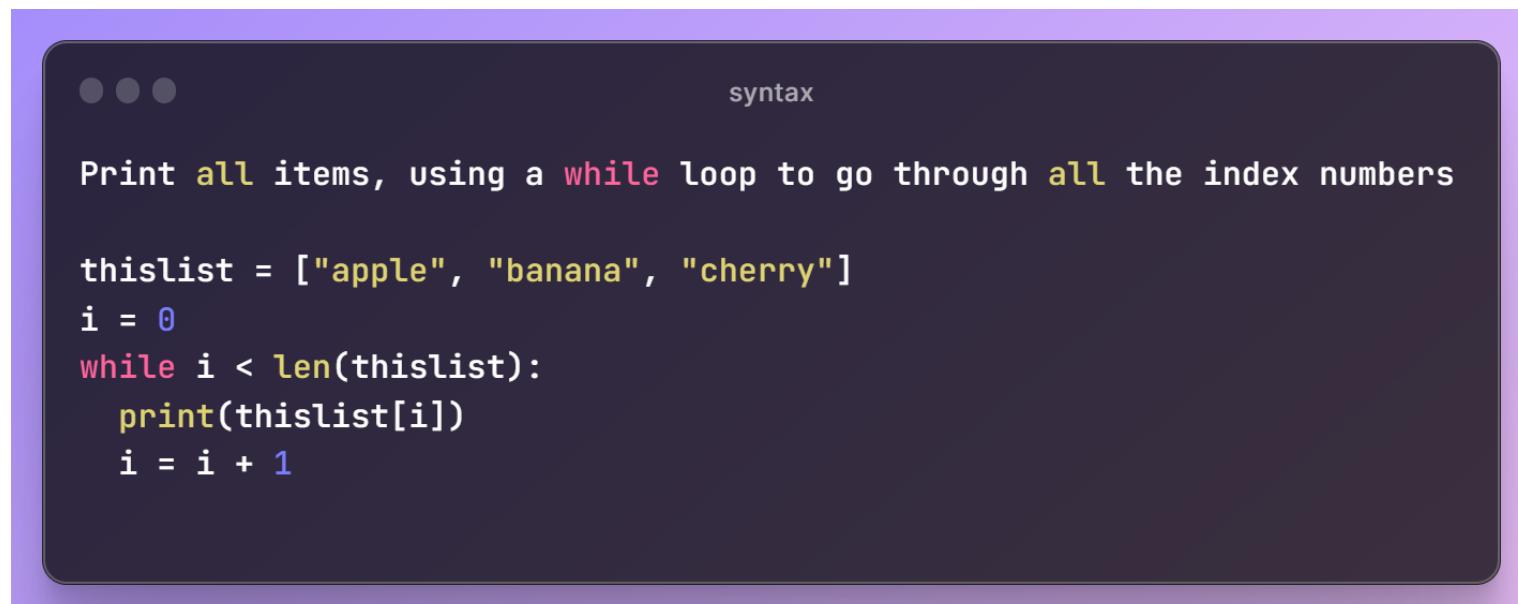
... syntax

Print `all` items by referring to their index number:

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

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.



A terminal window with a purple header bar containing three dots and the word "syntax". The main area of the terminal shows Python code. The code prints all items in a list using a while loop. The list contains "apple", "banana", and "cherry". The index variable starts at 0 and increases by 1 in each iteration until it reaches the length of the list.

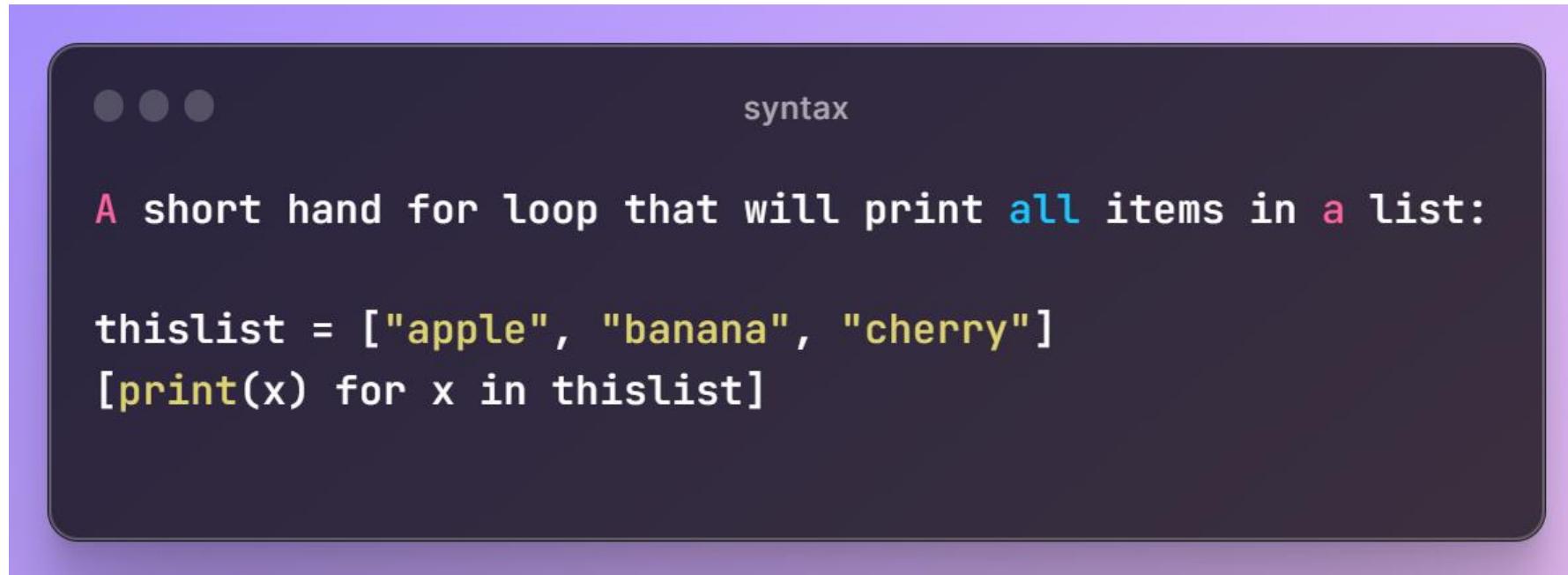
```
Print all items, using a while loop to go through all the index numbers

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

PYTHON

Looping Using List Comprehension

- List Comprehension offers the shortest syntax for looping through lists:



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 fruits, you want a new list, containing only the fruits with the letter "a" in the name.
- Without list comprehension you will have to write a **for** statement with a conditional test inside:

```
... syntax

fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

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

print(newlist)
```

List Comprehension

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

```
... syntax  
  
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
  
newlist = [x for x in fruits if "a" in x]  
  
print(newlist)  
print(newlist)
```

The Syntax

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

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

PYTHON

Condition

- The *condition* is like a filter that only accepts the items that evaluate to **True**.
- The condition `if x != "apple"` will return **True** for all elements other than "apple", making the new list contain all fruits except "apple".
- The *condition* is optional and can be omitted:

Only accept items that are not "apple":
`newlist = [x for x in fruits if x != "apple"]`

With no `if` statement:
`newlist = [x for x in fruits]`

PYTHON

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:



syntax

Set all values in the new list to 'hello':

```
newlist = ['hello' for x in fruits]
```



syntax

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

PYTHON

Expression

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



The *expression* in the example above says:

"Return the item if it is not banana, if it is banana return orange".

PYTHON

Sort List Alphanumerically

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

The image shows a screenshot of a Python code editor with two code snippets side-by-side.

Left Snippet:

- Header: syntax
- Text: Sort the list alphabetically:
- Code:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

Right Snippet:

- Header: syntax
- Text: Sort the list numerically:
- Code:

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

Sort Descending

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

...

syntax

Sort the list descending:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)
```

...

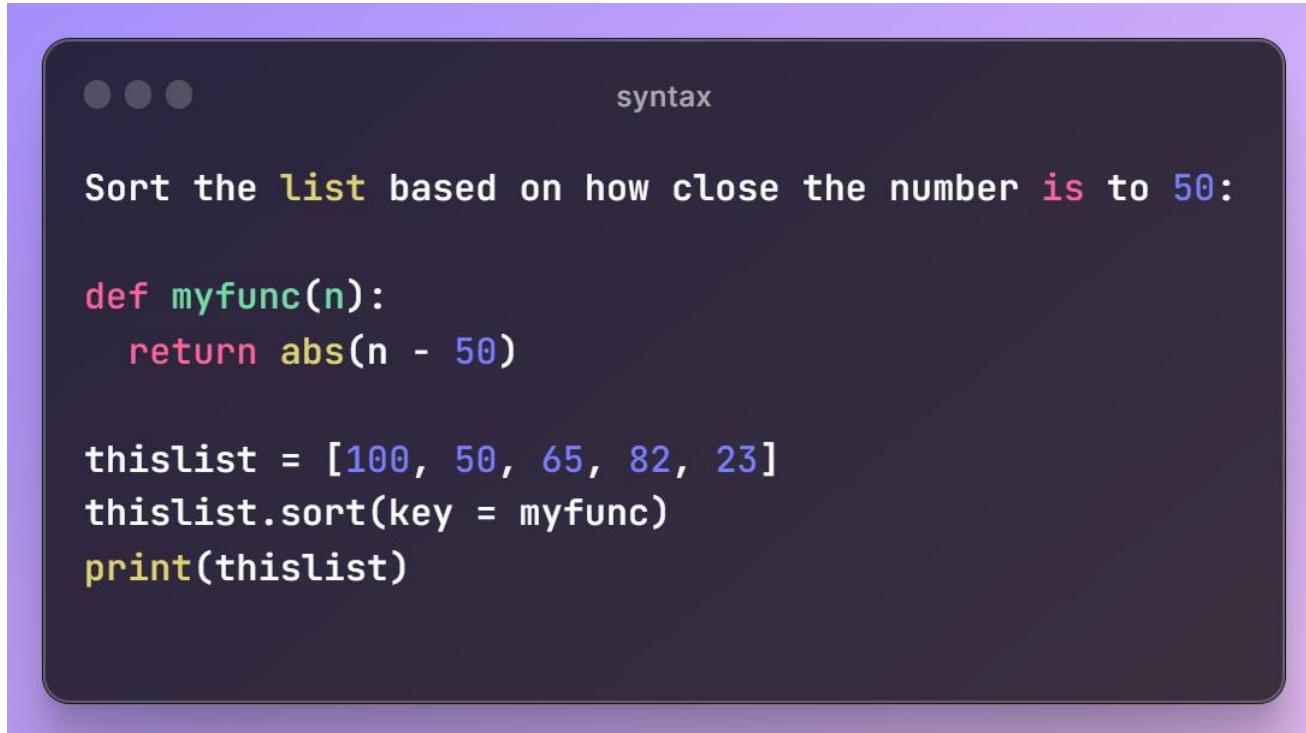
syntax

Sort the list descending:

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

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):



The image shows a dark-themed code editor window with a purple header bar. In the top right corner of the header bar, there are three small circular icons. To the right of these icons, the word "syntax" is written in a light gray font. The main body of the code editor is a dark gray rectangle with rounded corners. Inside this area, there is some white text representing Python code.

```
... syntax

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)
```

Case insensitive Sort

- By default the `sort()` method is case sensitive, resulting in all capital letters being sorted before lower case letters:
- Luckily we can use built-in functions as key functions when sorting a list.
- So if you want a case-insensitive sort function, use `str.lower` as a key function:



syntax

Case sensitive sorting can give an unexpected result:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)
```



syntax

Perform a case-insensitive sort of the list:

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)
```

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.

The screenshot shows a dark-themed code editor window with a purple header bar. In the header bar, there are three small circular icons on the left and the word "syntax" on the right. The main area of the editor contains the following text:

```
Reverse the order of the list items:  
  
thislist = ["banana", "Orange", "Kiwi", "cherry"]  
thislist.reverse()  
print(thislist)
```

PYTHON

Copy 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()`.

... syntax

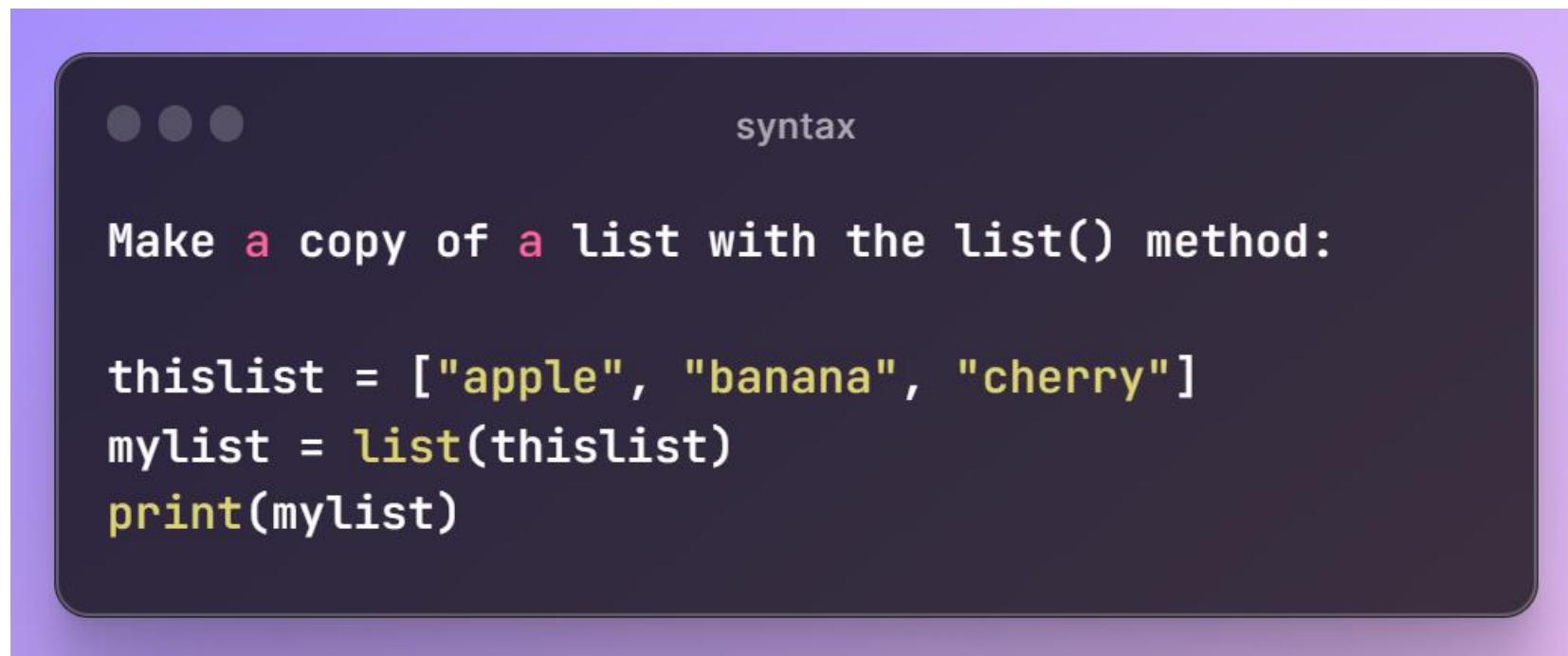
Make a copy of a list with the `copy()` method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

PYTHON

Copy List

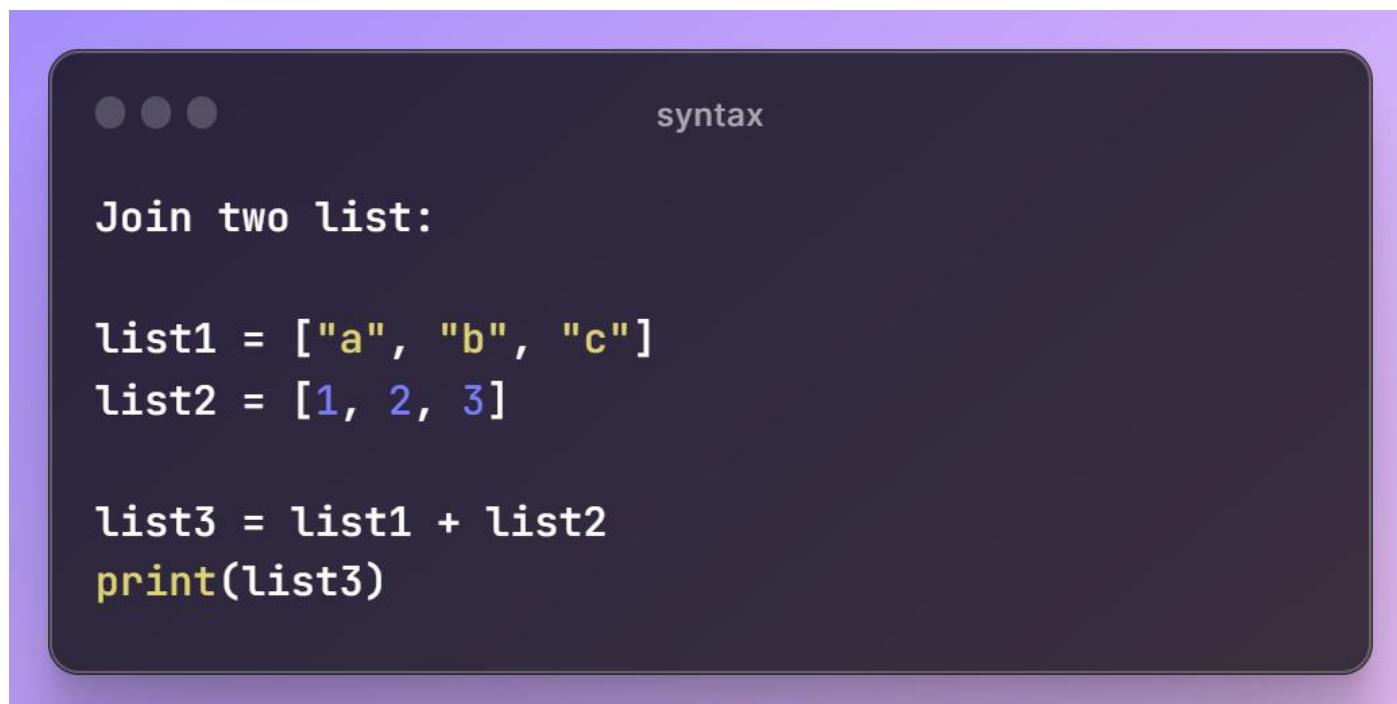
- Another way to make a copy is to use the built-in method `list()`.



PYTHON

Join Two List

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



Join two list:

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

list3 = list1 + list2
print(list3)
```

syntax

Join Two List

- Another way to join two lists is by appending all the items from list2 into list1, one by one:
- Or you can use the **extend()** method, where the purpose is to add elements from one list to another list:

Append list2 into list1:

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

for x in list2:
    list1.append(x)

print(list1)
```

syntax

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)
```

PYTHON

List methods

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

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

PYTHON

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.



The image shows a dark-themed code editor window with a light purple header bar. In the top right corner of the header bar, the word "syntax" is displayed. The main area of the editor contains three gray dots on the left, followed by the text "Create a Tuple:" in white. Below this, there is a code snippet in yellow and green:
`thistuple = ("apple", "banana", "cherry")
print(thistuple)`

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:



syntax

Tuples allow duplicate values:

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

Tuple Length

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

```
... syntax  
Print the number of items in the tuple:  
  
thistuple = ("apple", "banana", "cherry")  
print(len(thistuple))
```

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.

```
... syntax  
One item tuple, remember the comma:  
  
thistuple = ("apple",)  
print(type(thistuple))  
  
#NOT a tuple  
thistuple = ("apple")  
print(type(thistuple))
```

PYTHON

Tuple Item – Data Types

- Tuple items can be of any data type:
- A tuple can contain different data types:

... syntax

```
String, int and boolean data types:  
  
tuple1 = ("apple", "banana", "cherry")  
tuple2 = (1, 5, 7, 9, 3)  
tuple3 = (True, False, False)
```

... syntax

```
A tuple with strings, integers and boolean values:
```

```
tuple1 = ("abc", 34, True, 40, "male")
```

PYTHON

type()

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

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



The tuple() Constructor

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

Using the `tuple()` method to make a tuple:

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

Access Tuple Item

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



Note: The first item has index 0.

Negative Indexing

- Negative indexing means start from the end.
- -1 refers to the last item, -2 refers to the second last item etc.



syntax

Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

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.



Note: The search will start at index 2 (included) and end at index 5 (not included). Remember that the first item has index 0.

Range of Indexes

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

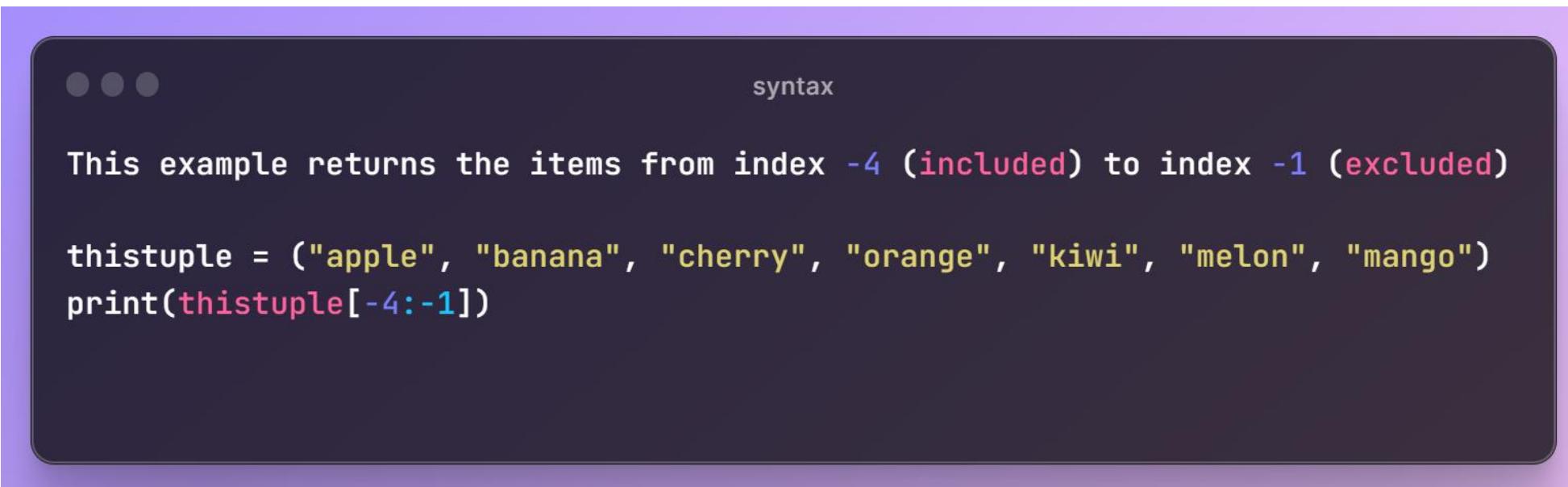
```
...  
syntax  
This example returns the items from the beginning to, but NOT included, "kiwi":  
  
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[:4])
```

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

```
...  
syntax  
This example returns the items from "cherry" and to the end:  
  
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[2:])
```

Range of Negative Indexes

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



The screenshot shows a terminal window with a dark background and light-colored text. At the top left, there are three small circular icons. To the right of them, the word "syntax" is written in a smaller font. Below this, a descriptive sentence is printed in white: "This example returns the items from index -4 (included) to index -1 (excluded)". Following this text, a line of Python code is shown in white: "thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango") print(thistuple[-4:-1])". The code defines a tuple named "thistuple" containing seven fruit names and then prints the slice from index -4 to -1.

PYTHON

Check If Item Exist

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



syntax

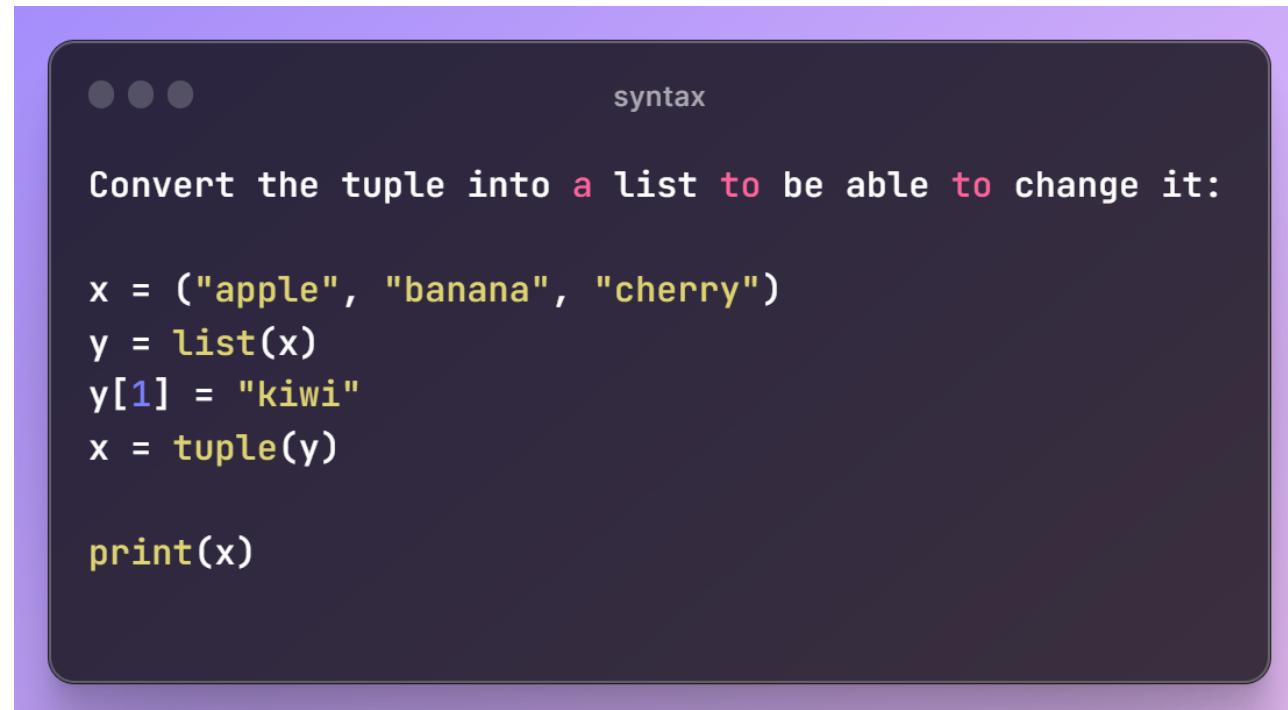
```
Check if "apple" is present in the tuple:  
  
thistuple = ("apple", "banana", "cherry")  
if "apple" in thistuple:  
    print("Yes, 'apple' is in the fruits tuple")
```

Update Tuples

- Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.
- But there are some workarounds.

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.



```
... syntax

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

x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

PYTHON

Add items

- Since tuples are immutable, they do not have a built-in `append()` method, but there are other ways to add items to a tuple.
 - 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.



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

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

Add items

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:

Note: When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.



The screenshot shows a dark-themed code editor window with a purple header bar. The header bar has three dots on the left and the word "syntax" on the right. The main area of the editor contains the following Python code:

```
...  
Create a new tuple with the value "orange", and add that tuple:  
  
thistuple = ("apple", "banana", "cherry")  
y = ("orange",)  
thistuple += y  
  
print(thistuple)
```

Remove items

Note: You cannot remove items in a tuple.

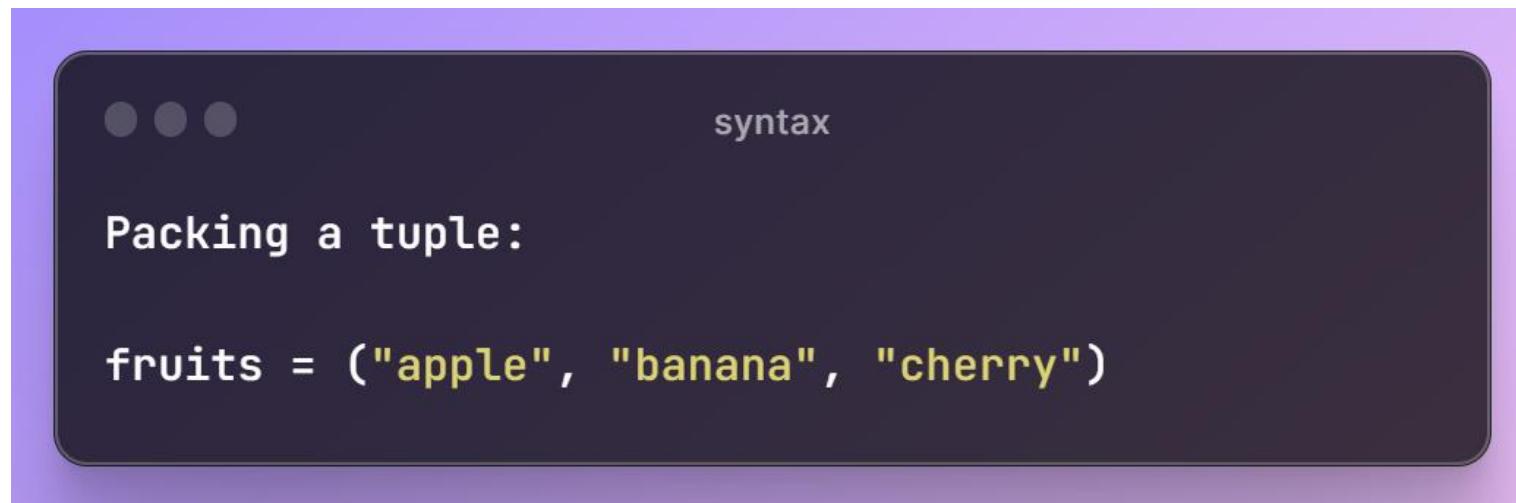
- Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

```
...  
syntax  
  
Convert the tuple into a list, remove "apple", and convert it back into a tuple:  
  
thistuple = ("apple", "banana", "cherry")  
y = list(thistuple)  
y.remove("apple")  
thistuple = tuple(y)
```

```
...  
syntax  
  
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
```

Packing a Tuple

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



Unpacking a Tuple

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

Note: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.



• • • syntax

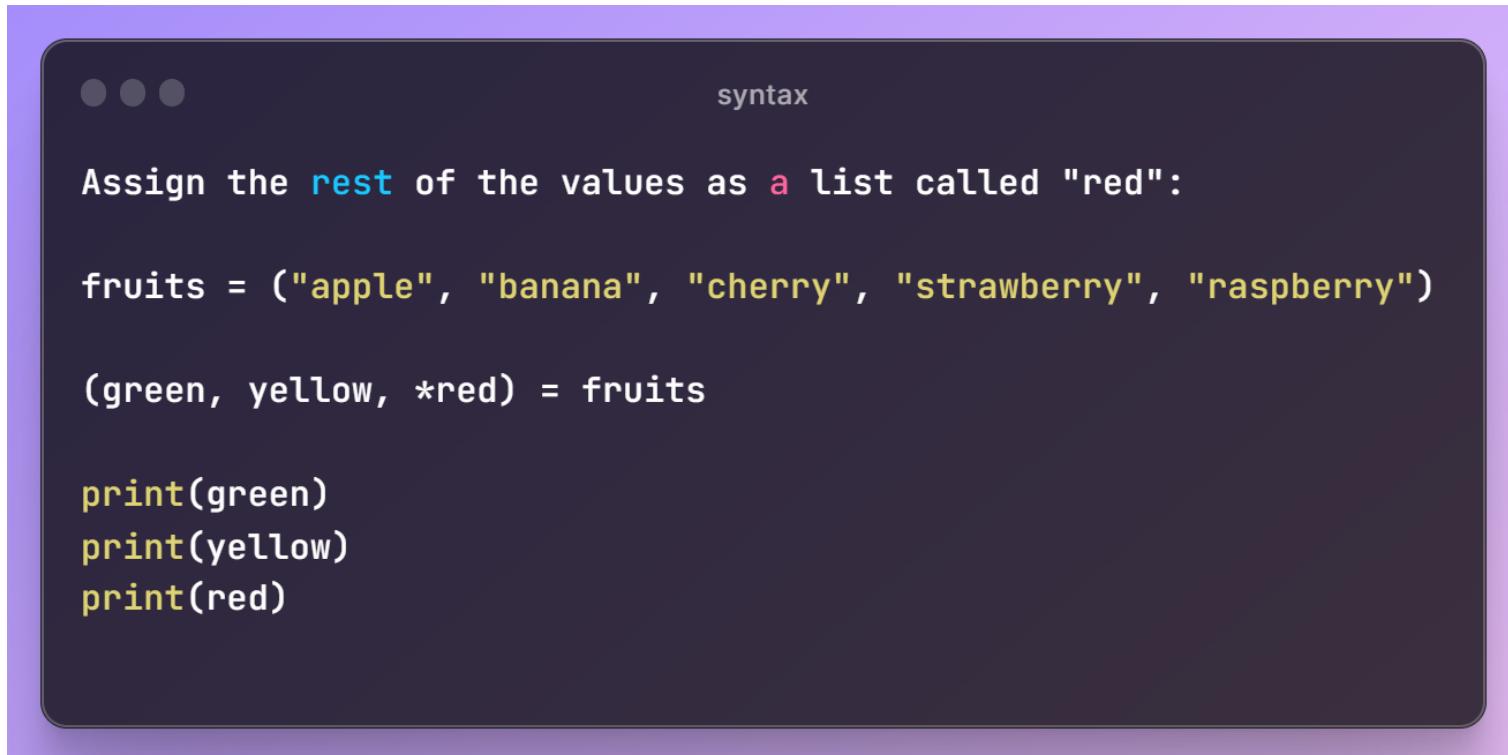
Unpacking a tuple:

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

PYTHON

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:



The screenshot shows a dark-themed code editor window with a light purple header bar. The header bar has three dots on the left and the word "syntax" on the right. The main code area contains the following Python code:

```
... syntax

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

fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

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

PYTHON

Using Asterisk*

- 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.



The screenshot shows a Jupyter Notebook cell with the following content:

```
... syntax  
Add a list of values the "tropic" variable:  
  
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")  
  
(green, *tropic, red) = fruits  
  
print(green)  
print(tropic)  
print(red)
```

Loop Through a Tuple

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



syntax

Iterate through the items and **print** the values:

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

Loop Through the Index Numbers

- You can also loop through the tuple items by referring to their index number.
- Use the `range()` and `len()` functions to create a suitable iterable.



PYTHON

Using a While Loop

- You can loop through the tuple 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 after each iteration.

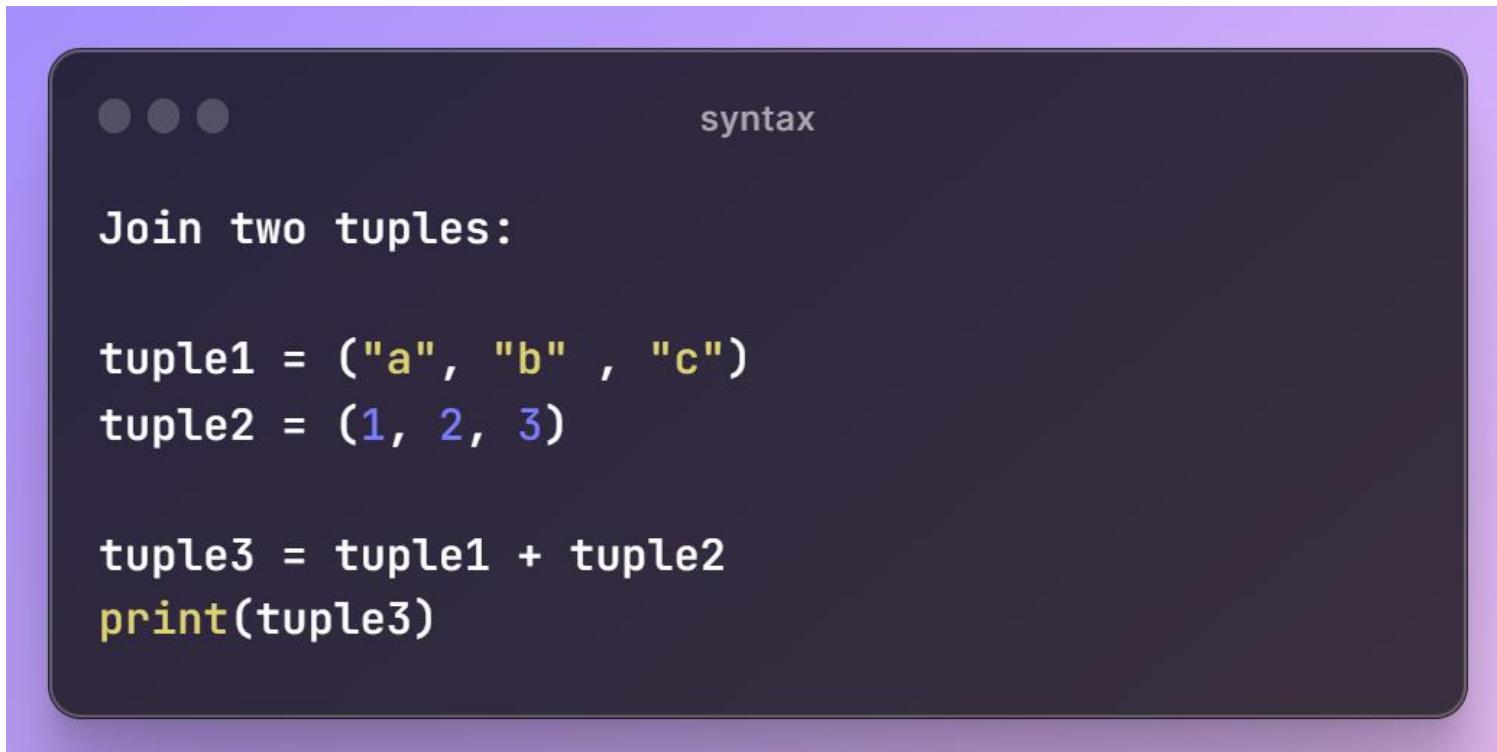
syntax

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
```

Join Two Tuples

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



The image shows a dark-themed code editor window with a purple gradient background. At the top left, there are three gray circular icons. To the right of them, the word "syntax" is written in white. The main area of the window contains the following Python code:

```
Join two tuples:

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

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

Multiply Tuples

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

The code block features a dark background with a purple header and footer. In the top-left corner of the header, there are three small white dots. To the right of the dots, the word "syntax" is written in a light color. The main content area contains the following text:
Multiply the fruits tuple by 2:

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

print(mytuple)`

PYTHON

Tuple Method

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

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

PYTHON

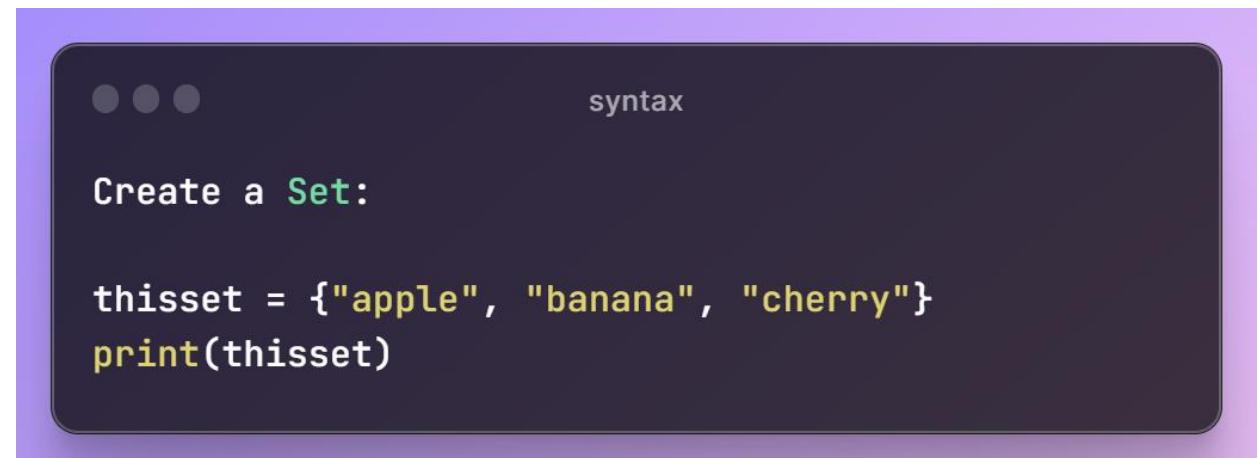
Sets

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

- 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*.

Note: Set *items* are unchangeable, but you can remove items and add new items.

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

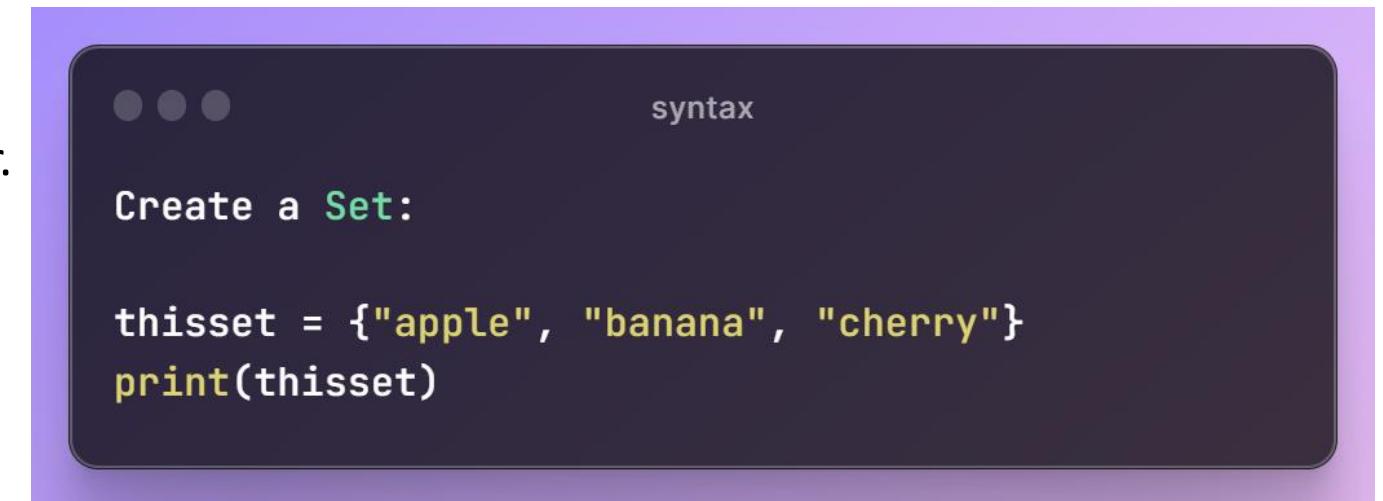


PYTHON

Sets

* **Note:** Set *items* are unchangeable, but you can remove items and add new items.

Note: Sets are unordered, so you cannot be sure in which order the items will appear.



Set Items

- Set items are unordered, unchangeable, and do not allow duplicate values

Sets

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.
- Once a set is created, you cannot change its items, but you can remove items and add new items.

Duplicates Not Allowed

- Sets cannot have two items with the same value.

Note: The values `True` and `1` are considered the same value in sets, and are treated as duplicates:



syntax

Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}
```

```
print(thisset)
```

PYTHON

Sets

Note: The values **False** and **0** are considered the same value in sets, and are treated as duplicates:



syntax

True and **1** is considered the same value:

```
thisset = {"apple", "banana", "cherry", True, 1, 2}
```

```
print(thisset)
```



syntax

False and **0** is considered the same value:

```
thisset = {"apple", "banana", "cherry", False, True, 0}
```

```
print(thisset)
```

PYTHON

Sets

- Get the Length of a Set
- To determine how many items a set has, use the `len()` function.



PYTHON

Sets Item – Data Type

- Set items can be of any data type:

... syntax
String, int and boolean data types:

set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}

- A set can contain different data types:

... syntax
A set with strings, integers and boolean values:

set1 = {"abc", 34, True, 40, "male"}

PYTHON

type()

- From Python's perspective, sets are defined as objects with the data type 'set':
`<class 'set'>`



PYTHON

The `set()` Constructor

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



syntax

Using the `set()` constructor to make a set:

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

Access Set-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.

```
...           syntax  
Loop through the set, and print the values:  
  
thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
    print(x)
```

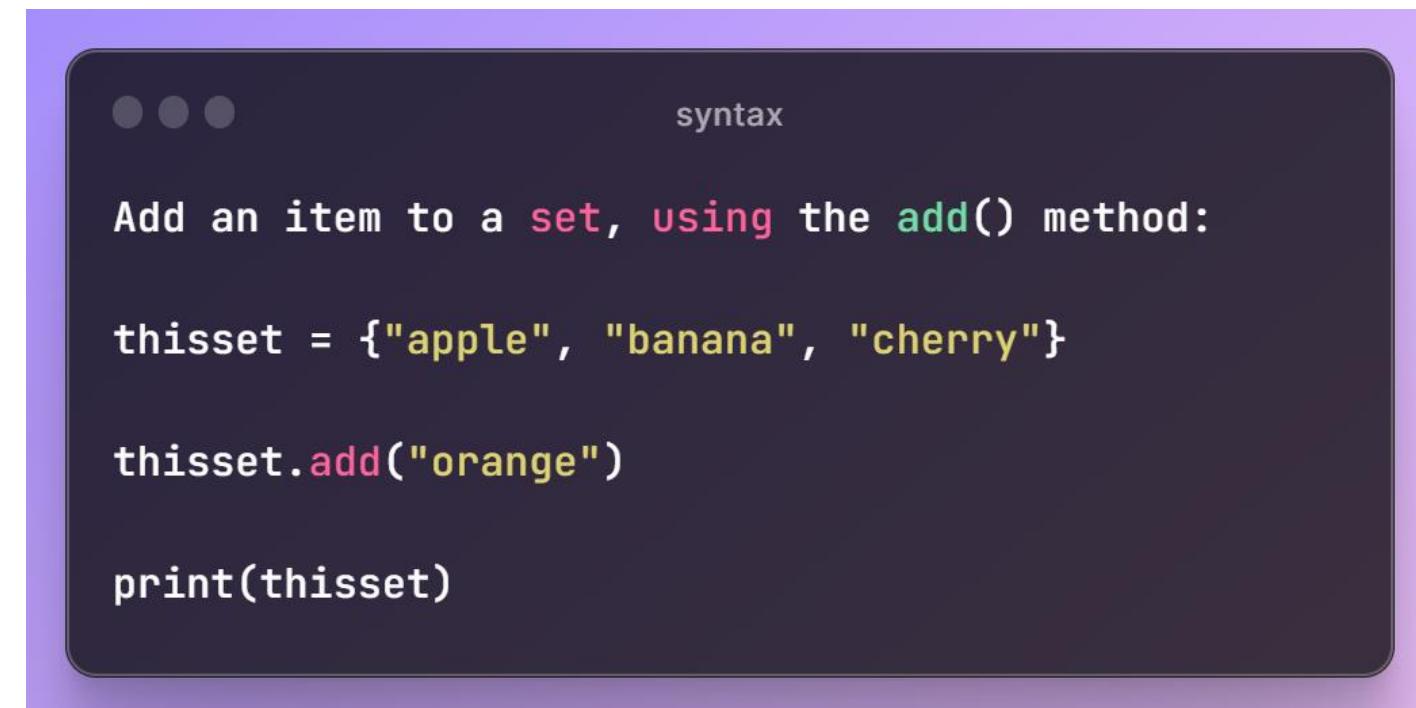
```
...           syntax  
Check if "banana" is present in the set:  
  
thisset = {"apple", "banana", "cherry"}  
  
print("banana" in thisset)
```

Change Items

- Once a set is created, you cannot change its items, but you can add new items.

Add Set Items

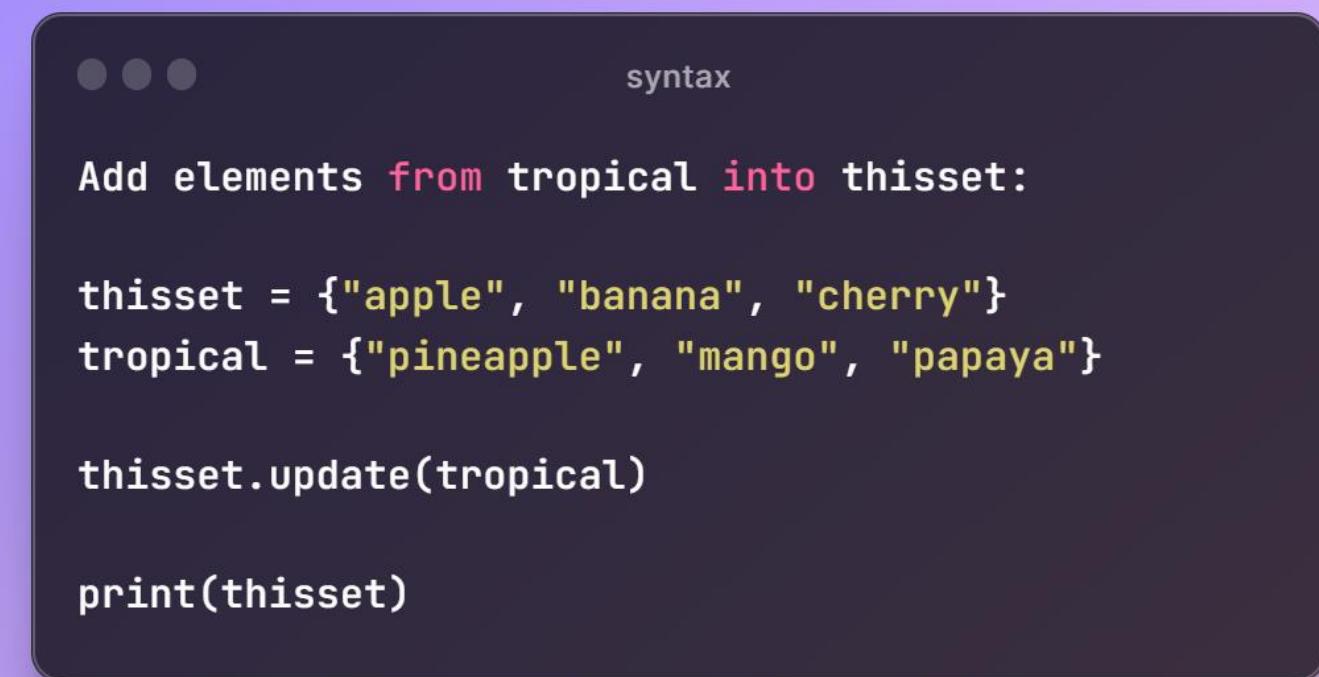
- Once a set is created, you cannot change its items, but you can add new items.
- To add one item to a set use the `add()` method.



PYTHON

Add Set

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

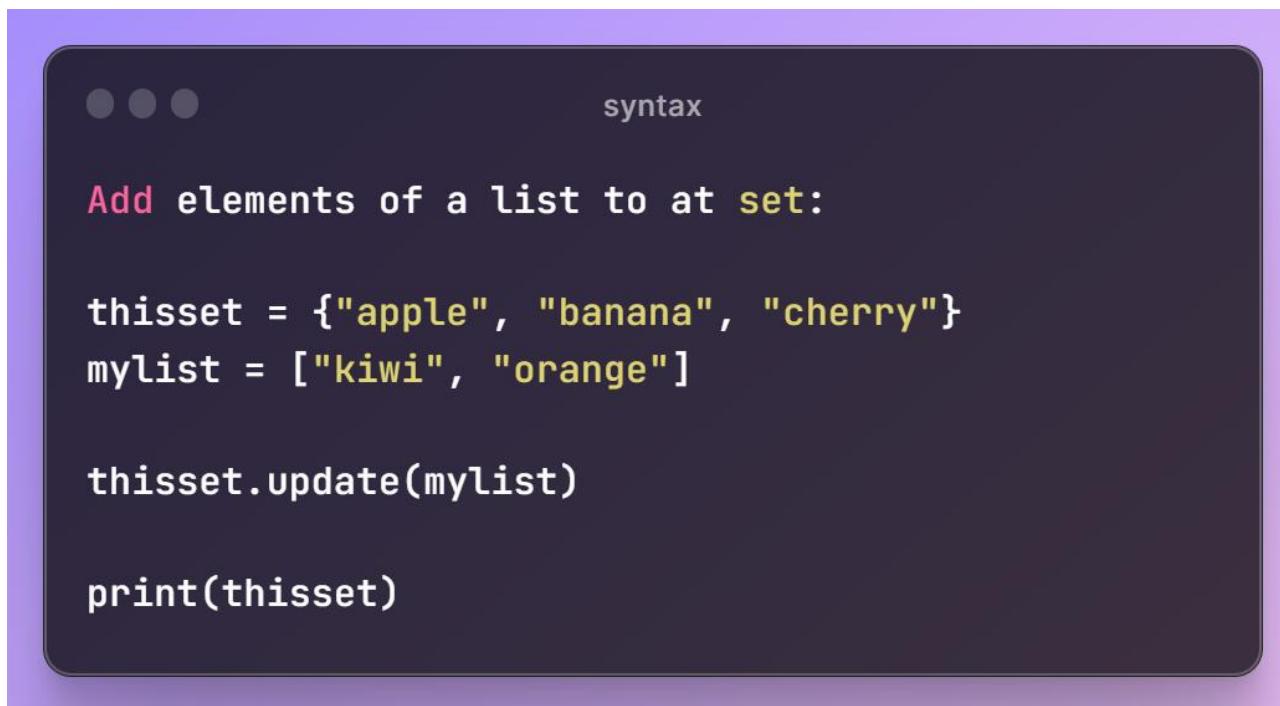


syntax

```
Add elements from tropical into thisset:  
  
thisset = {"apple", "banana", "cherry"}  
tropical = {"pineapple", "mango", "papaya"}  
  
thisset.update(tropical)  
  
print(thisset)
```

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.).



A screenshot of a dark-themed code editor window. In the top right corner, there are three small circular icons. To the right of them, the word "syntax" is written in a light gray font. The main area of the window contains the following Python code:

```
...  
syntax  
  
Add elements of a list to at set:  
  
thisset = {"apple", "banana", "cherry"}  
mylist = ["kiwi", "orange"]  
  
thisset.update(mylist)  
  
print(thisset)
```

Remove Set item

- To remove an item in a set, use the `remove()`, or the `discard()` method.

...

syntax

```
Remove "banana" by using the remove() method:  
  
thisset = {"apple", "banana", "cherry"}  
  
thisset.remove("banana")  
  
print(thisset)
```

...

syntax

```
Remove "banana" by using the discard() method:  
  
thisset = {"apple", "banana", "cherry"}  
  
thisset.discard("banana")  
  
print(thisset)
```

Note: If the item to remove does not exist, `remove()` will raise an error.

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

Remove Set item

- You can also use the `pop()` method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.
- The return value of the `pop()` method is the removed item.

Note: Sets are *unordered*, so when using the `pop()` method, you do not know which item that gets removed.



syntax

Remove a random item by using the `pop()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
x = thisset.pop()  
  
print(x)  
  
print(thisset)
```

Remove Set item



syntax

The `clear()` method empties the set:

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



syntax

The `del` keyword will delete the set completely:

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

PYTHON

Loop Set

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



syntax

Loop through the set, and print the values:

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

Join Set

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:

```
... syntax  
The union() method returns a new set with all items from both sets:  
  
set1 = {"a", "b" , "c"}  
set2 = {1, 2, 3}  
  
set3 = set1.union(set2)  
print(set3)
```

```
... syntax  
The update() method inserts the items in set2 into set1:  
  
set1 = {"a", "b" , "c"}  
set2 = {1, 2, 3}  
  
set1.update(set2)  
print(set1)
```

Note: Both **union()** and **update()** will exclude any duplicate items.

Keep Only the Duplicate

- The `intersection_update()` method will keep only the items that are present in both sets.
- The `intersection()` method will return a *new* set, that only contains the items that are present in both sets.



syntax

Keep 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)
```



syntax

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

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

PYTHON

Keep All , But NOT the Duplicate

- The `symmetric_difference_update()` method will keep only the elements that are NOT present in both sets.



The screenshot shows a dark-themed code editor window. At the top left, there are three small circular icons. To the right of them, the word "syntax" is displayed. The main area contains the following text:

```
Keep the items that are not present in both sets:  
  
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}  
  
x.symmetric_difference_update(y)  
  
print(x)
```

Keep All , But NOT the Duplicate

- The `symmetric_difference()` method will return a new set, that contains only the elements that are NOT present in both sets.



syntax

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

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

PYTHON

Keep All , But NOT the Duplicate

Note: The values **True** and **1** are considered the same value in sets, and are treated as duplicates:



The image shows a screenshot of a Jupyter Notebook cell. The cell contains the following code:

```
... syntax  
True and 1 is considered the same value:  
  
x = {"apple", "banana", "cherry", True}  
y = {"google", 1, "apple", 2}  
  
z = x.symmetric_difference(y)  
  
print(z)
```

The code demonstrates the use of the `symmetric_difference` method on sets. It creates two sets, `x` and `y`, where `x` contains unique strings and `y` contains integers. The `symmetric_difference` method returns a new set containing all elements that are in either `x` or `y`, but not in both. The output of the code is printed at the bottom of the cell.

PYTHON

Set Methods

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

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all the elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns a set containing the difference between two or more sets
<code>difference_update()</code>	Removes the items in this set that are also included in another, specified set
<code>discard()</code>	Remove the specified item
<code>intersection()</code>	Returns a set, that is the intersection of two other sets
<code>intersection_update()</code>	Removes the items in this set that are not present in other, specified set(s)
<code>isdisjoint()</code>	Returns whether two sets have a intersection or not
<code>issubset()</code>	Returns whether another set contains this set or not
<code>issuperset()</code>	Returns whether this set contains another set or not

PYTHON

Set Methods

<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

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:



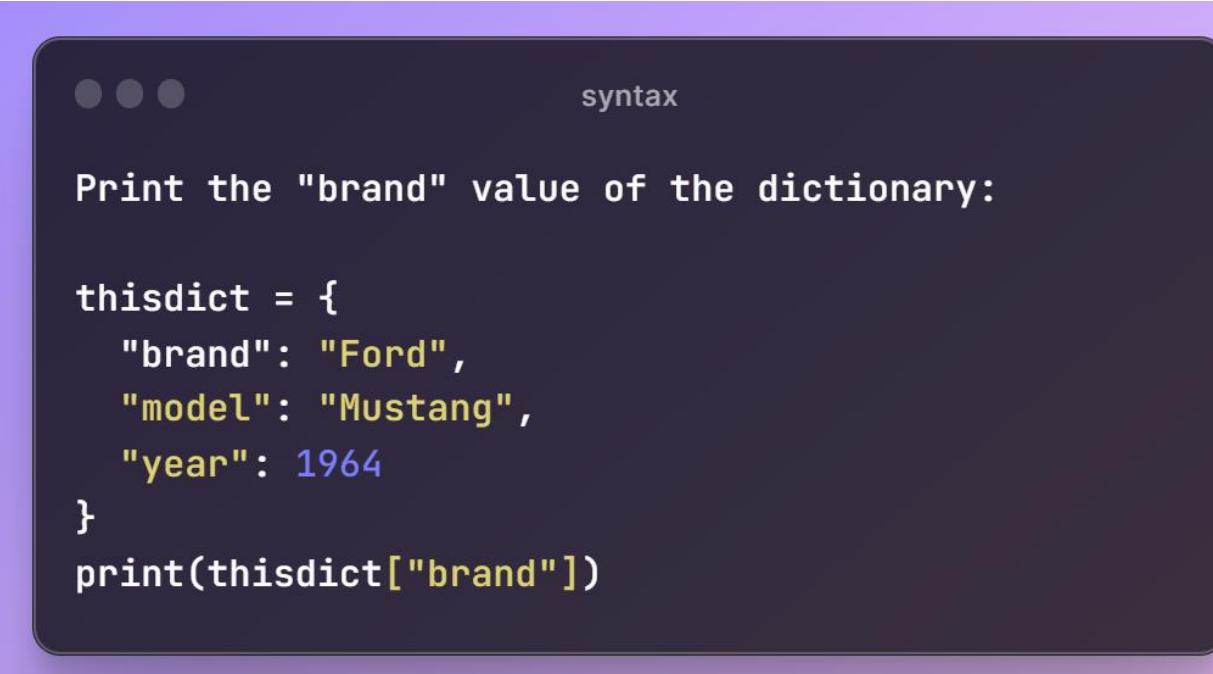
syntax

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.



The image shows a screenshot of a Python code editor. At the top left, there are three small circular icons. To the right of them, the word "syntax" is written in a light blue font. Below this, the text "Print the 'brand' value of the dictionary:" is displayed in white. Underneath, a Python code snippet is shown in white text on a dark background:

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

Ordered or Unordered ?

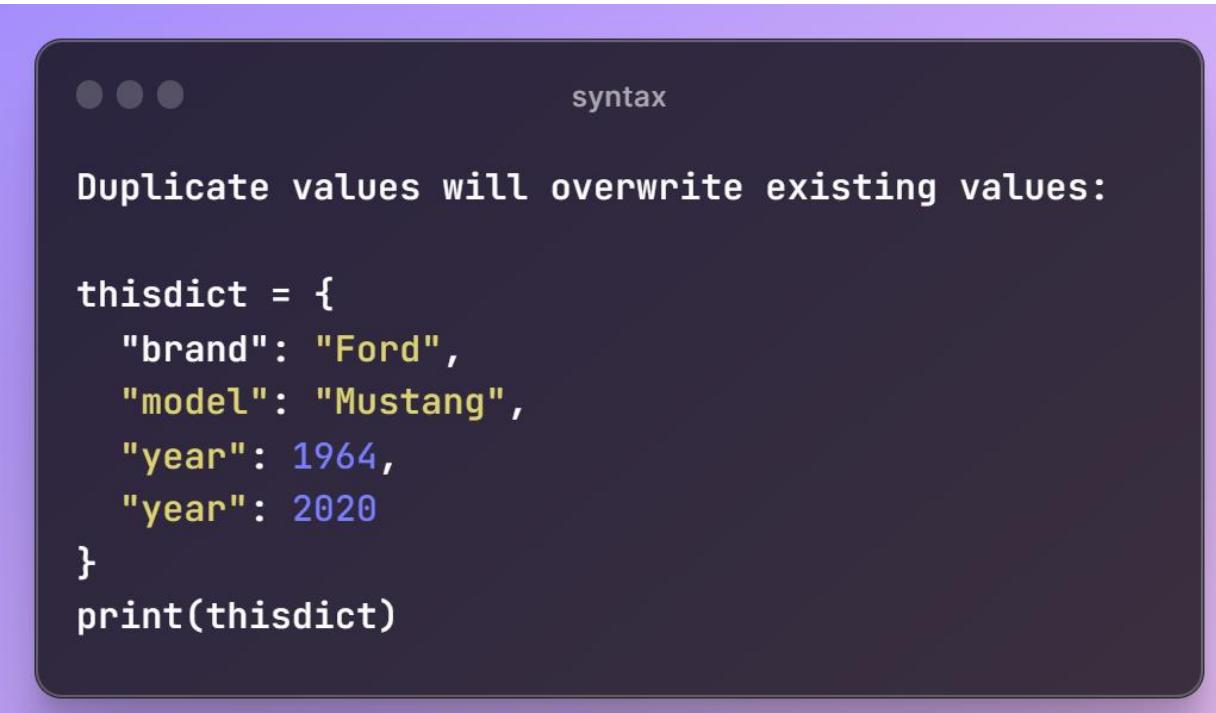
- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
- Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

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:



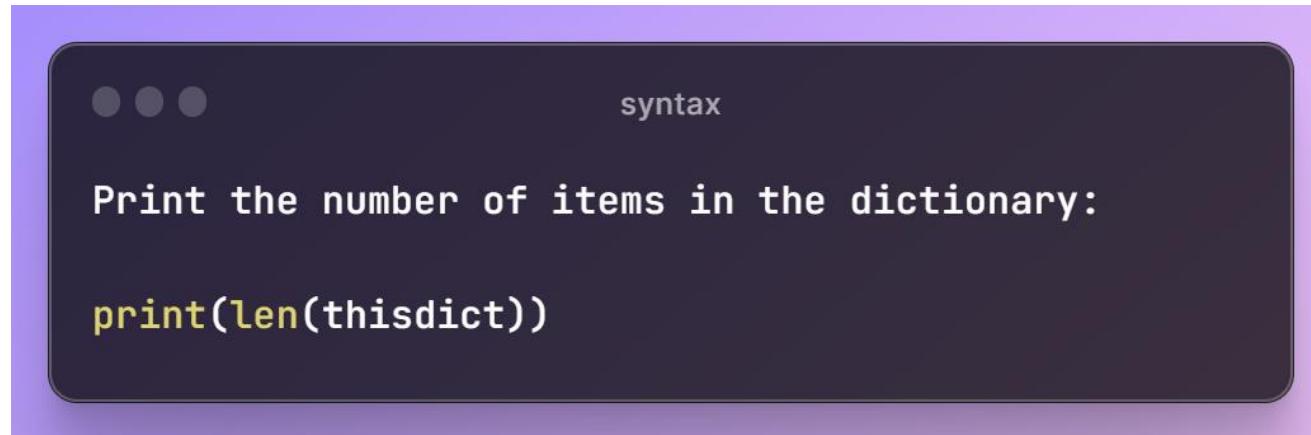
The code editor window shows the following Python code:

```
...syntax  
  
Duplicate values will overwrite existing values:  
  
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964,  
    "year": 2020  
}  
print(thisdict)
```

The code demonstrates that attempting to assign a new value to an existing key ("year": 2020) overwrites the previous value (1964). The code is highlighted with syntax colors: purple for the class name, blue for the function name, and green for the string values.

Dictionary Length

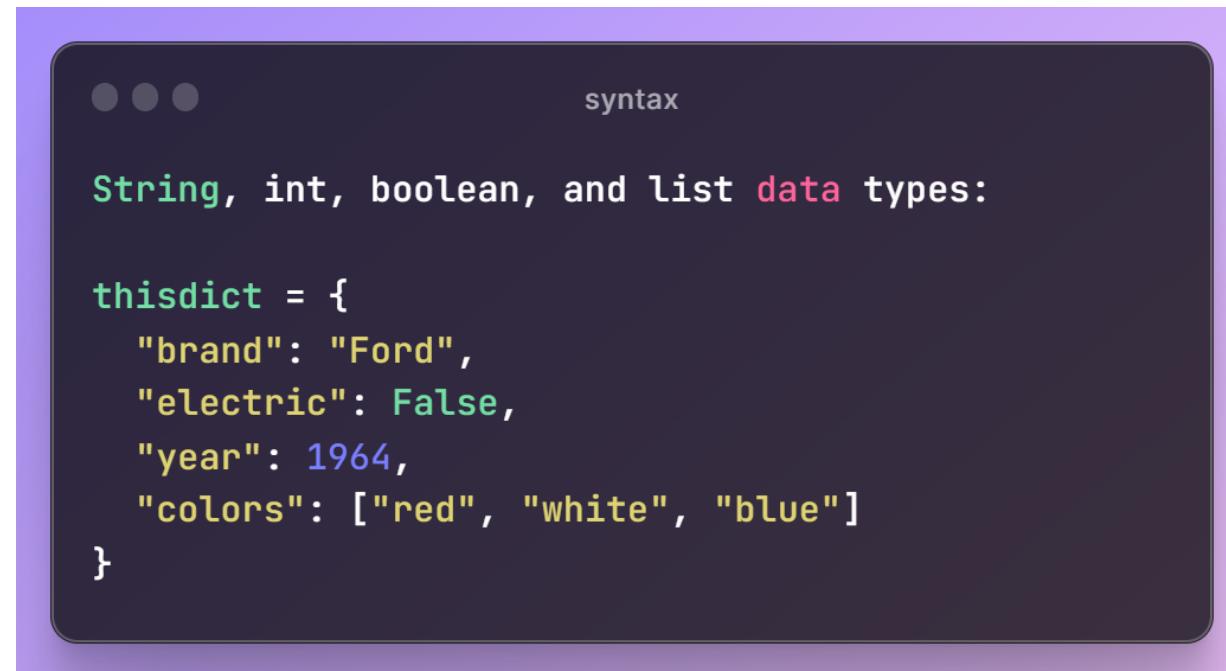
- To determine how many items a dictionary has, use the `len()` function:



PYTHON

Dictionary Items – Data Types

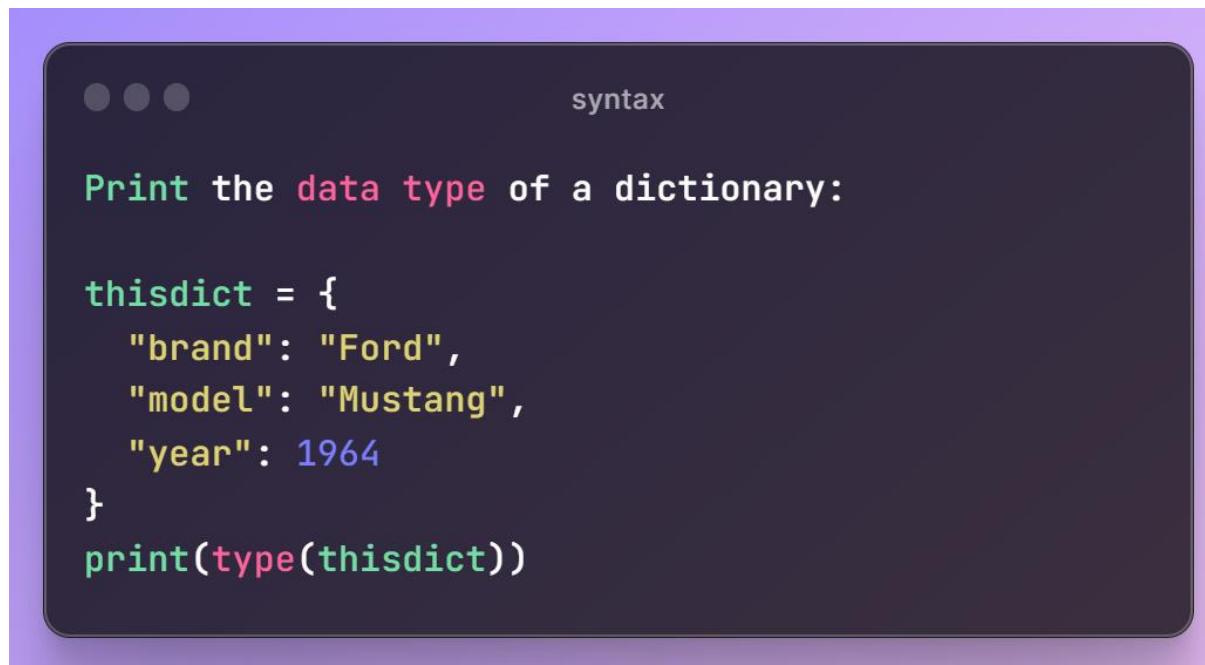
- The values in dictionary items can be of any data type:



PYTHON

type()

- From Python's perspective, dictionaries are defined as objects with the data type 'dict':
`<class 'dict'>`



The image shows a screenshot of a Python code editor. The code is as follows:

```
... syntax

Print the data type of a dictionary:

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

PYTHON

The dict() Constructor

- It is also possible to use the `dict()` constructor to make a dictionary.

• • •

syntax

Using the `dict()` method to make a dictionary:

```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

Access Dictionary item

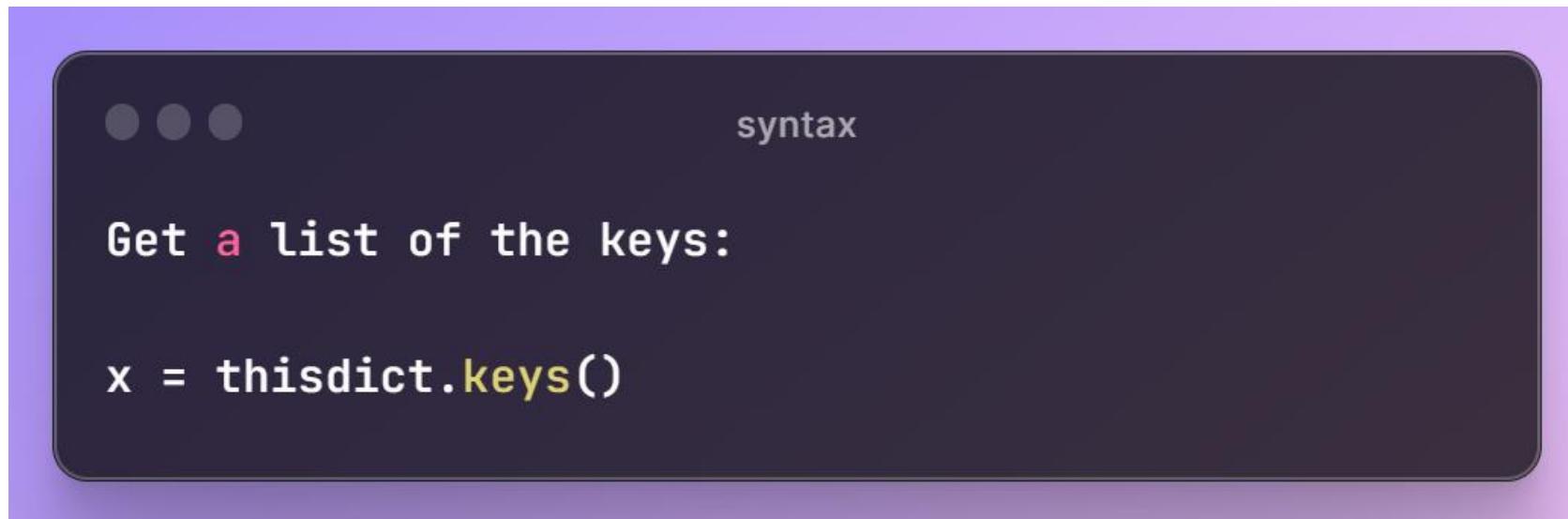
- You can access the items of a dictionary by referring to its key name, inside square brackets:
- There is also a method called `get()` that will give you the same result:

```
... syntax  
Get the value of the "model" key:  
  
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
x = thisdict["model"]
```

```
... syntax  
Get the value of the "model" key:  
  
x = thisdict.get("model")
```

Get Keys

- The `keys()` method will return a list of all the keys in the dictionary.
- The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.



PYTHON

Get Keys

• • •

syntax

Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.keys()  
  
print(x) #before the change  
  
car["color"] = "white"  
  
print(x) #after the change
```

Get Values

- The **values()** method will return a list of all the values in the dictionary.
- The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

...
...

syntax

Get a list of the values:

```
x = thisdict.values()
```

Get Values

```
● ● ●                                     syntax

Make a change in the original dictionary, and see that the values list gets updated as well:

car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}

x = car.values()

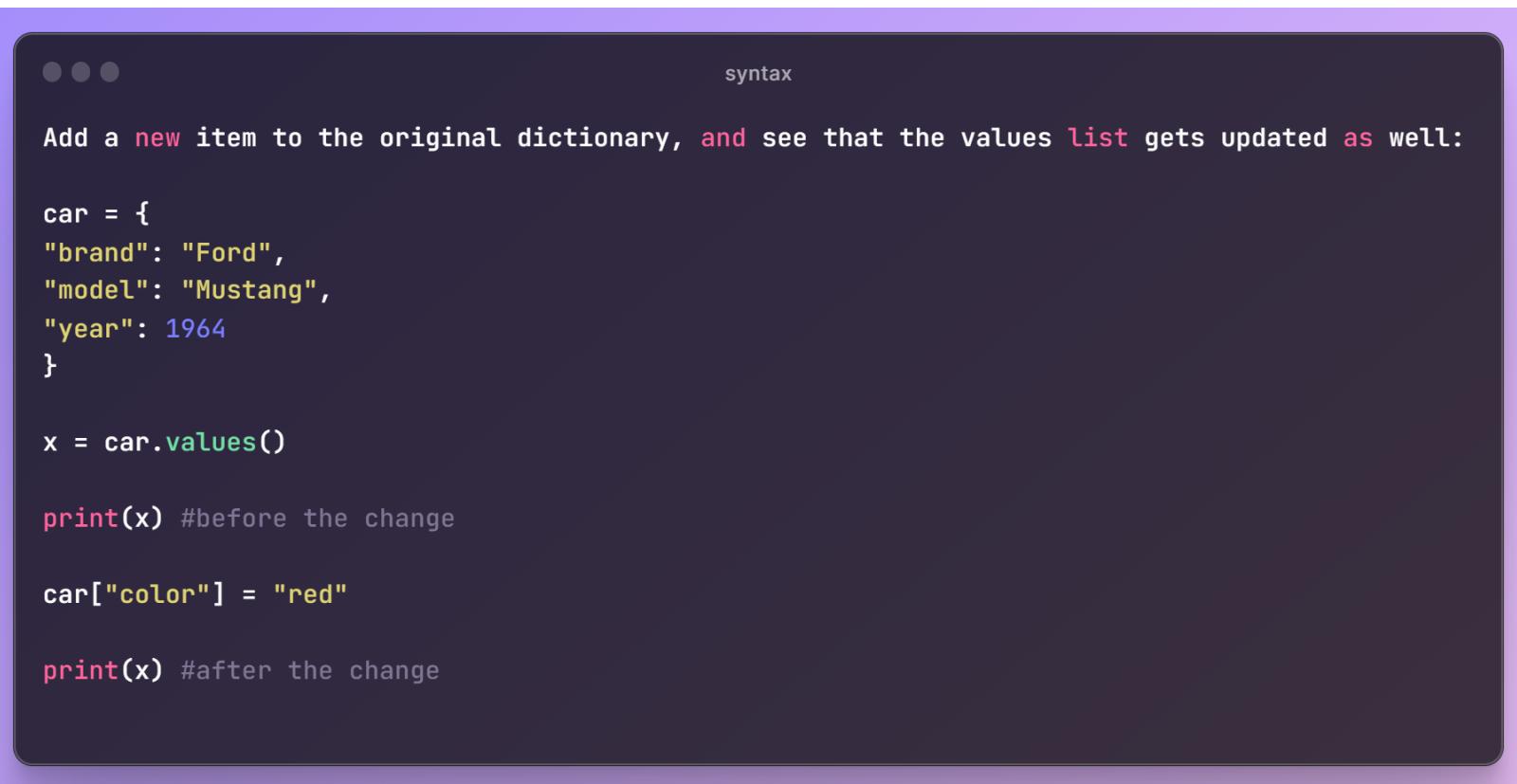
print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

PYTHON

Get Values



The image shows a screenshot of a Jupyter Notebook cell. At the top left, there are three gray dots indicating multiple lines of code. On the right, the word "syntax" is displayed. The main content of the cell is a Python script. It starts with a variable assignment `car = {` followed by four key-value pairs: `"brand": "Ford"`, `"model": "Mustang"`, `"year": 1964`, and a closing brace `}`. Below this, the code continues with `x = car.values()`, `print(x) #before the change`, `car["color"] = "red"`, and `print(x) #after the change`. The code is color-coded: `car` and `x` are blue, while the print statements and the new value are pink.

```
... syntax

Add a new item to the original dictionary, and see that the values list gets updated as well:

car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

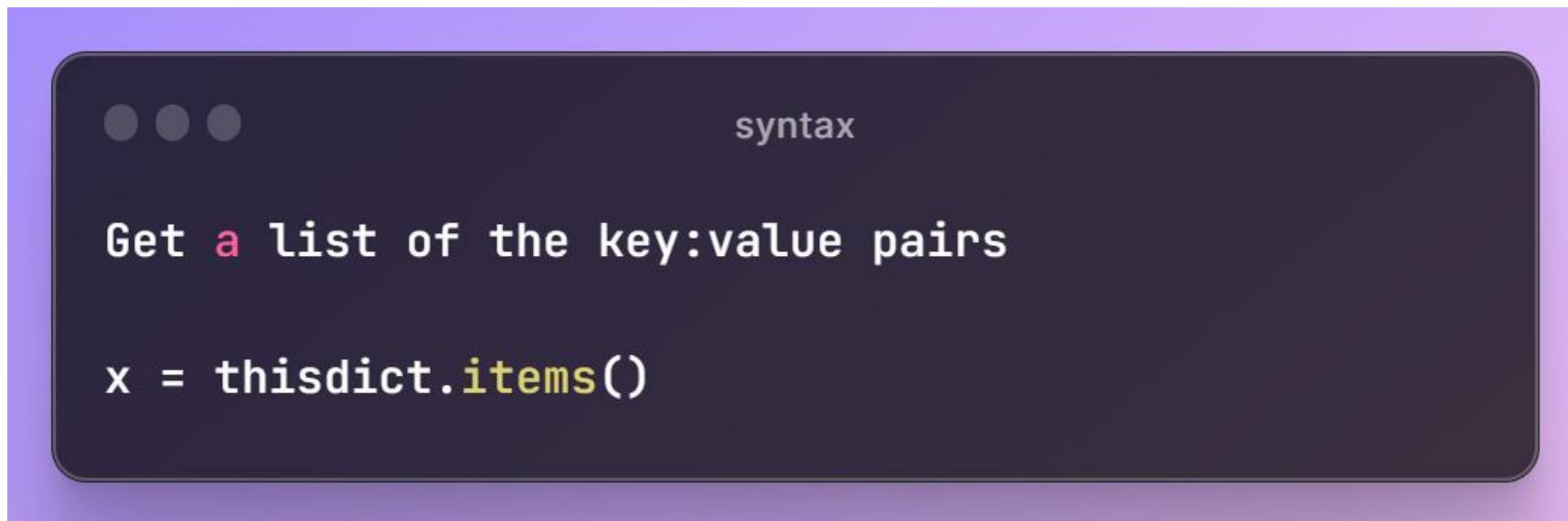
print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

Get Items

- The `items()` method will return each item in a dictionary, as tuples in a list.
- The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.



PYTHON

Get Items

```
... syntax  
Make a change in the original dictionary, and see that the items list gets updated as well:  
  
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.items()  
  
print(x) #before the change  
  
car["year"] = 2020  
  
print(x) #after the change
```

PYTHON

Get Items

• • •

syntax

Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
x = car.items()  
  
print(x) #before the change  
  
car["color"] = "red"  
  
print(x) #after the change
```

Check if Key Exists

- To determine if a specified key is present in a dictionary use the **in** keyword:

• • •

syntax

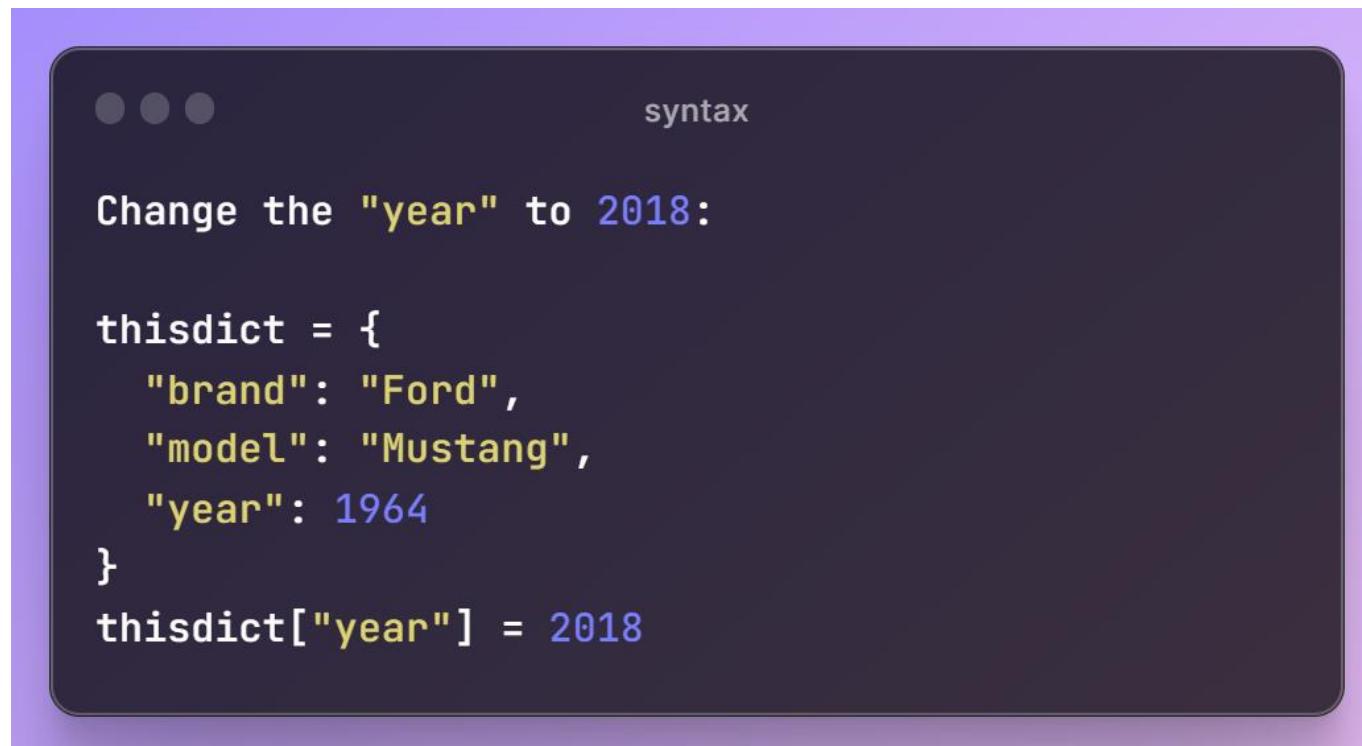
Check if "model" is present in the dictionary:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
if "model" in thisdict:  
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Change Dictionary Items

Change Values

- You can change the value of a specific item by referring to its key name:



The image shows a screenshot of a Python code editor. At the top, there are three circular icons followed by the word "syntax". Below this, the text "Change the "year" to 2018:" is displayed. Then, a Python dictionary is defined with the key "year" having the value 1964. Finally, the value of the "year" key is changed to 2018.

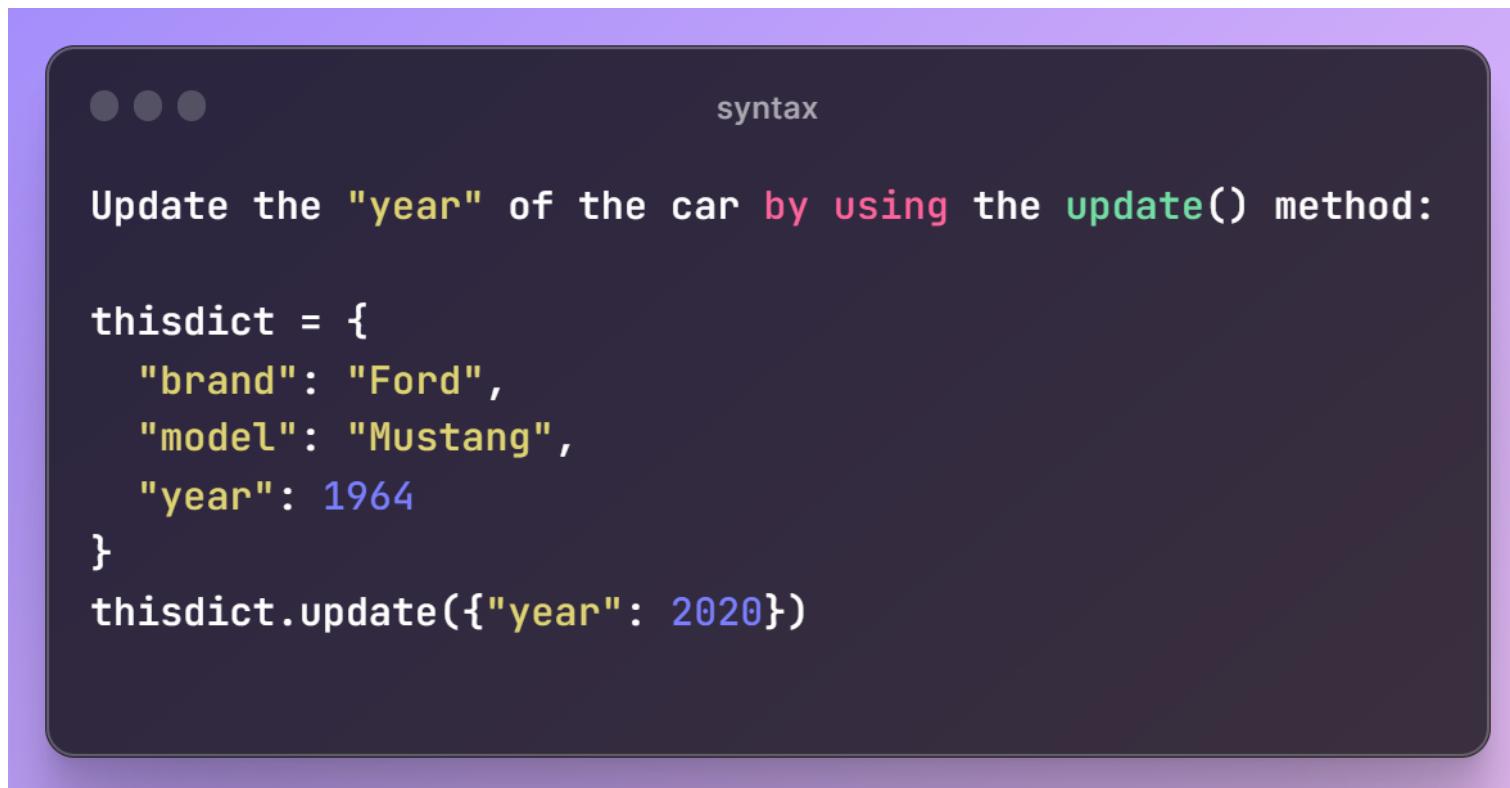
```
... syntax

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.



The screenshot shows a dark-themed code editor window. At the top left, there are three small circular icons. To the right of them, the word "syntax" is displayed. The main area contains the following code:

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

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

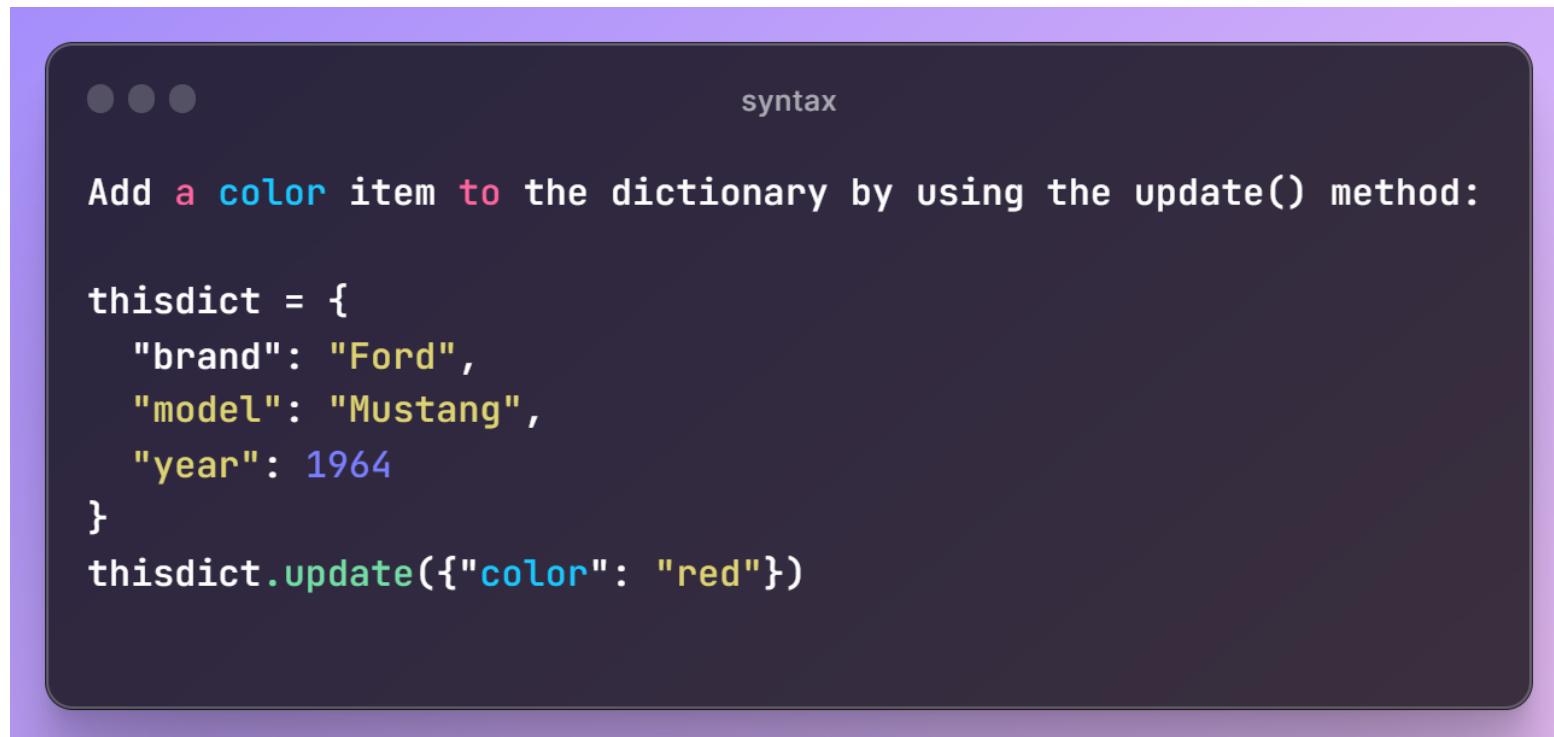
Add Dictionary Items

- Adding an item to the dictionary is done by using a new index key and assigning a value to it:

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

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.



The image shows a screenshot of a Python code editor with a dark theme. At the top left, there are three small circular icons. To the right of them, the word "syntax" is written in a light gray font. Below this, a line of text reads "Add a color item to the dictionary by using the update() method:". Underneath, there is a sample Python code snippet:

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

Removing Items

- There are several methods to remove items from a dictionary:

The screenshot shows a dark-themed code editor window. In the top right corner, there are three small circular icons. To their right, the word "syntax" is written in a light gray font. The main area of the editor contains the following Python code:

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

The code defines a dictionary named `thisdict` with three key-value pairs: `"brand": "Ford"`, `"model": "Mustang"`, and `"year": 1964`. It then uses the `pop()` method to remove the item with the key `"model"`. Finally, it prints the updated dictionary.

Removing Items

- There are several methods to remove items from a dictionary:



syntax

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)
```

Removing Items



syntax

The `del` keyword removes the item with the specified key name:

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

Removing Items



syntax

The `del` keyword can also `delete` the dictionary completely:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
del thisdict  
  
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Removing Items

• • •

syntax

The `clear()` method empties the dictionary:

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

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.

Print all key names in the dictionary, one by one:

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

Print all values in the dictionary, one by one:

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

Loop Through a Dictionary



syntax

You can also use the `values()` method to return values of a dictionary:

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

Loop Through a Dictionary



syntax

You can use the `keys()` method to return the keys of a dictionary:

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

Loop Through a Dictionary



syntax

Loop through both keys and values, by using the `items()` method:

```
for x, y in thisdict.items():
    print(x, y)
```

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()`.

• • •

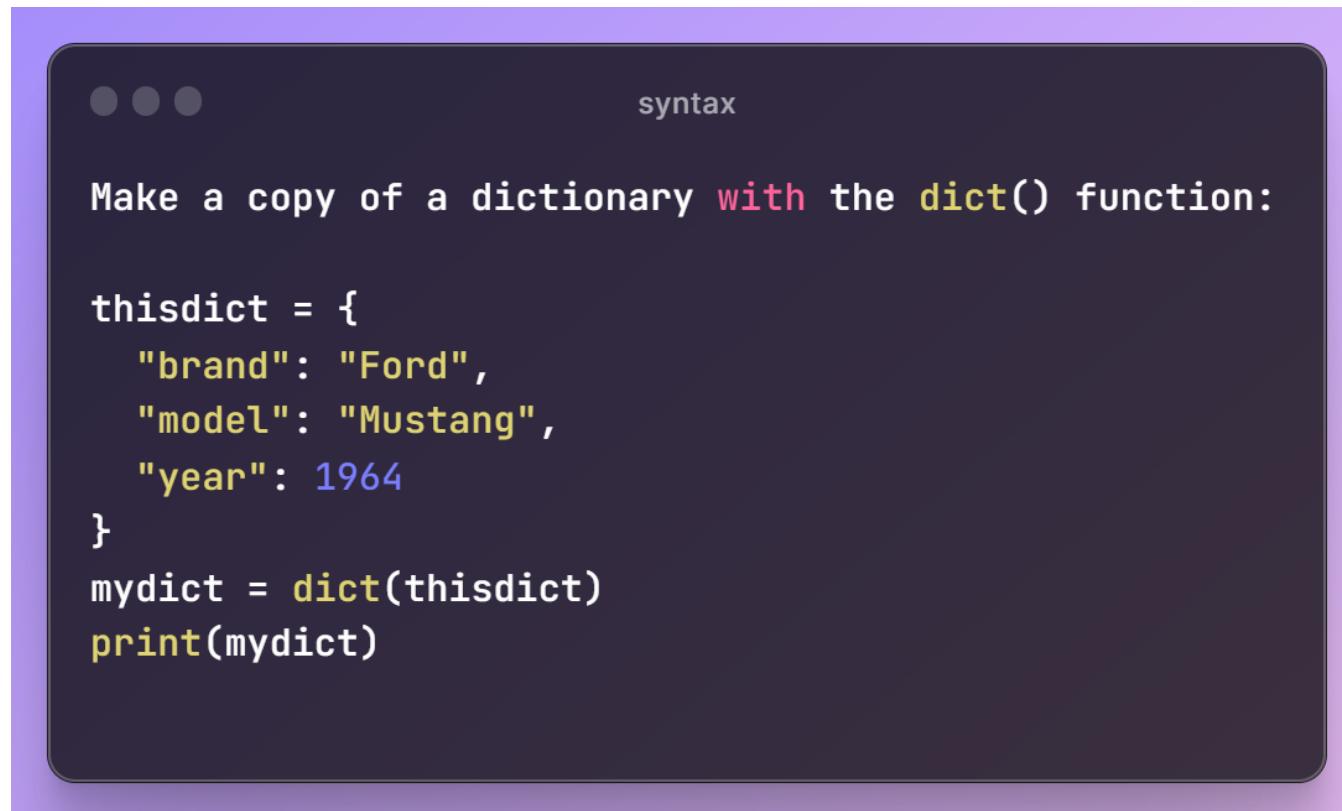
syntax

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

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

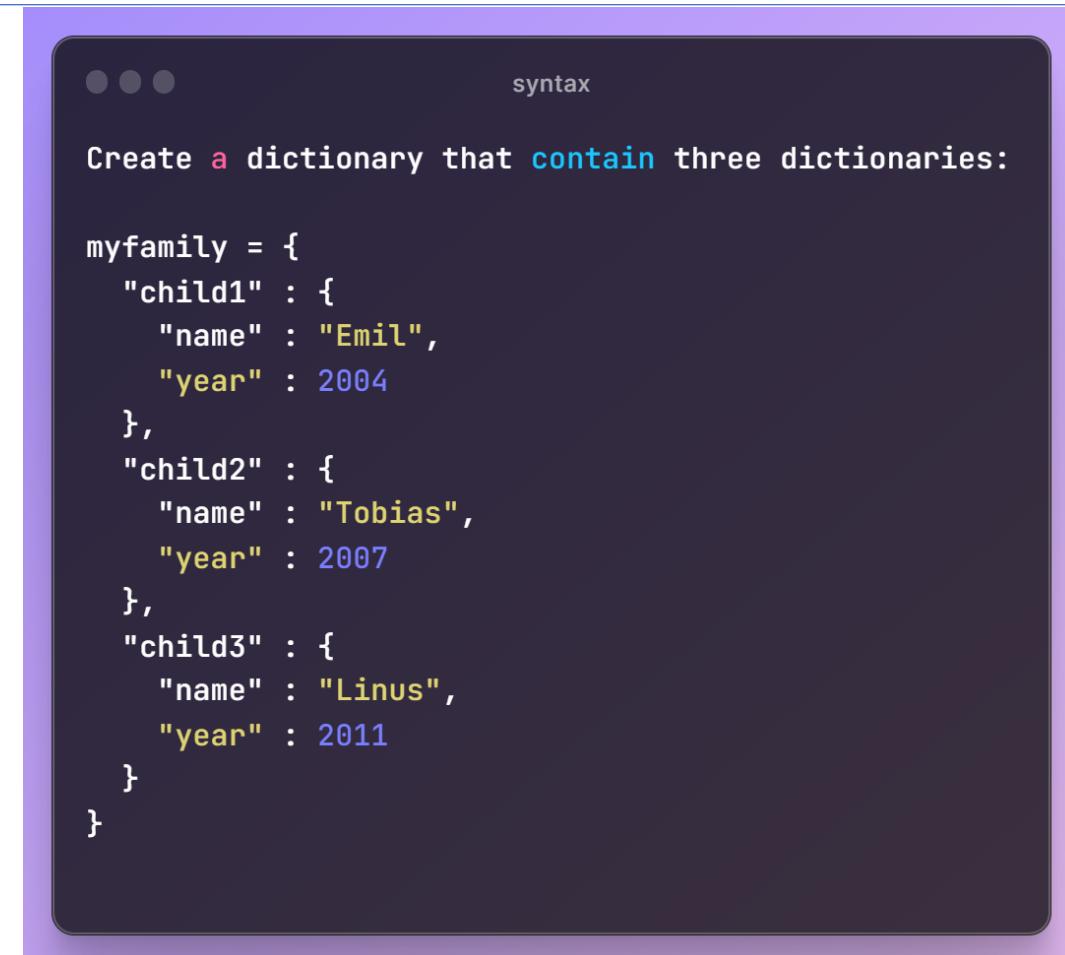
Copy a Dictionary

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



Nested Dictionary

- A dictionary can contain dictionaries, this is called **nested dictionaries**.



The image shows a dark-themed code editor window with a purple header bar. The header bar has three dots on the left and the word "syntax" on the right. The main area of the editor contains Python code demonstrating nested dictionaries. The code is as follows:

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

Nested Dictionary

- Or if you dictionary want to add three dictionaries into a new

```
...  
syntax  
  
Create three dictionaries, then create one dictionary that will contain the other three  
dictionaries:  
  
child1 = {  
    "name" : "Emil",  
    "year" : 2004  
}  
child2 = {  
    "name" : "Tobias",  
    "year" : 2007  
}  
child3 = {  
    "name" : "Linus",  
    "year" : 2011  
}  
  
myfamily = {  
    "child1" : child1,  
    "child2" : child2,  
    "child3" : child3  
}
```

Access Items in Nested Dictionaries

- To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary:



syntax

Print the name of child 2:

```
print(myfamily["child2"]["name"])
```

PYTHON

Dictionary Methods

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

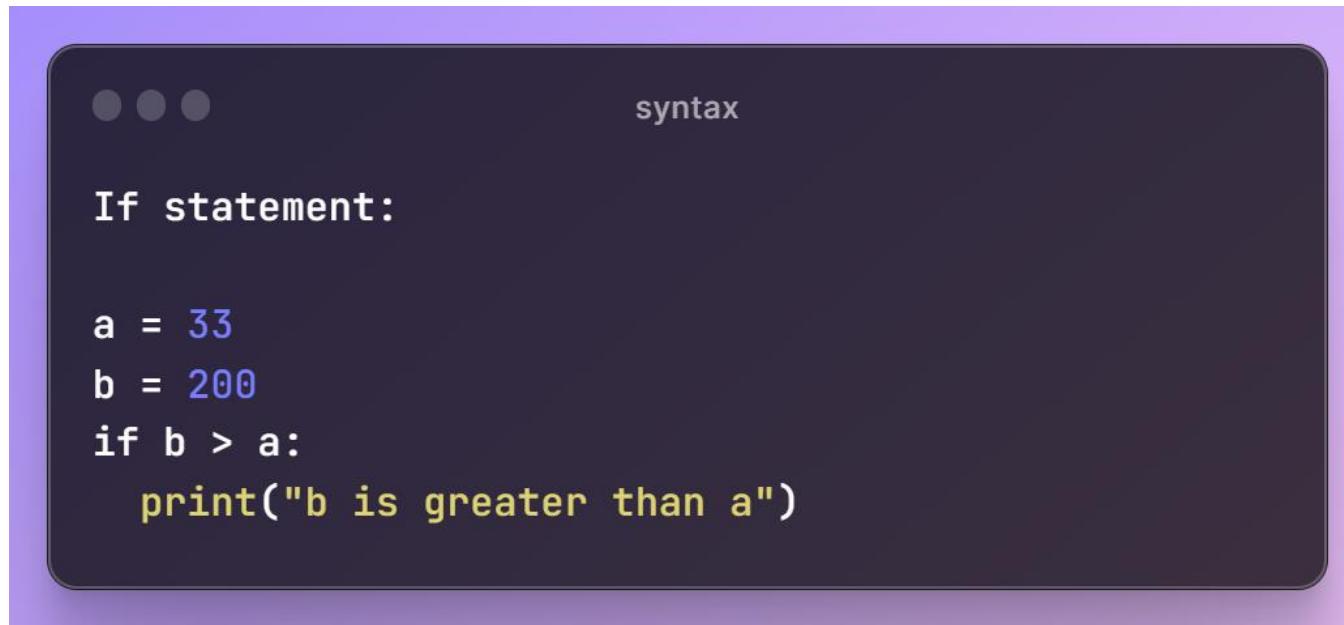
Method	Description
<code>clear()</code>	Removes all the elements from the dictionary
<code>copy()</code>	Returns a copy of the dictionary
<code>fromkeys()</code>	Returns a dictionary with the specified keys and value
<code>get()</code>	Returns the value of the specified key
<code>items()</code>	Returns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

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.

PYTHON

Conditions And If Statements



The image shows a dark-themed code editor window with a light purple header bar. In the top right corner of the header bar, there are three small gray dots. To the right of these dots, the word "syntax" is written in a smaller white font. The main area of the code editor contains the following Python code:

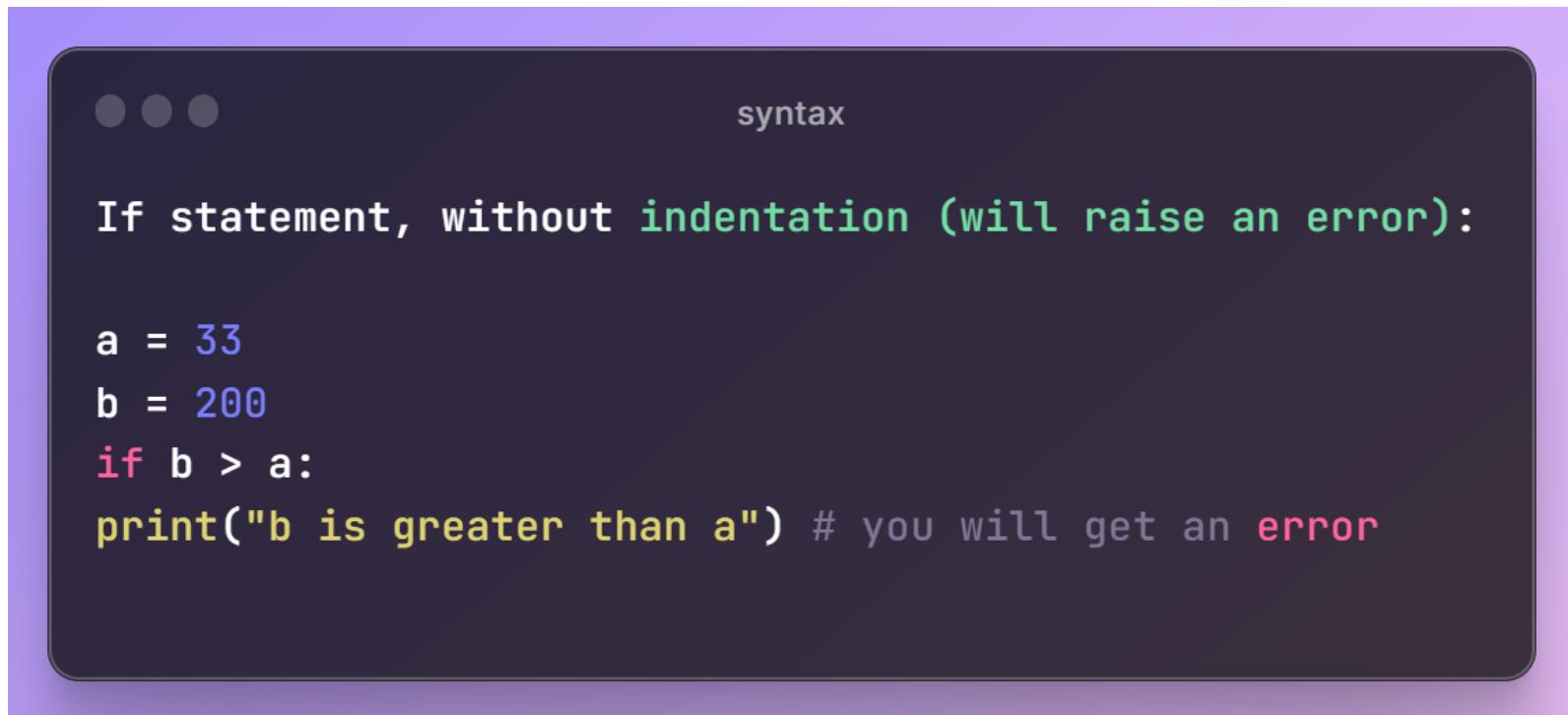
```
...  
If statement:  
  
a = 33  
b = 200  
if b > a:  
    print("b is greater than a")
```

- 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".

PYTHON

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.



The image shows a terminal window with a dark background and light-colored text. At the top left, there are three small circular icons. To the right of them, the word "syntax" is displayed. Below this, a message is shown: "If statement, without indentation (will raise an error):". Following this message, several lines of Python code are listed:
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error

PYTHON

Elif

The **elif** keyword is Python's way of saying "if the previous conditions were not true, then try this condition".

In this example **a** is equal to **b**, so the first condition is not true, but the **elif** condition is true, so we print to screen that "a and b are equal".

```
...  
syntax  
  
a = 33  
b = 33  
if b > a:  
    print("b is greater than a")  
elif a == b:  
    print("a and b are equal")
```

PYTHON

Else

The **else** keyword catches anything which isn't caught by the preceding conditions.

In this example **a** is greater than **b**, so the first condition is not true, also the **elif** condition is not true, so we go to the **else** condition and print to screen that "a is greater than b".

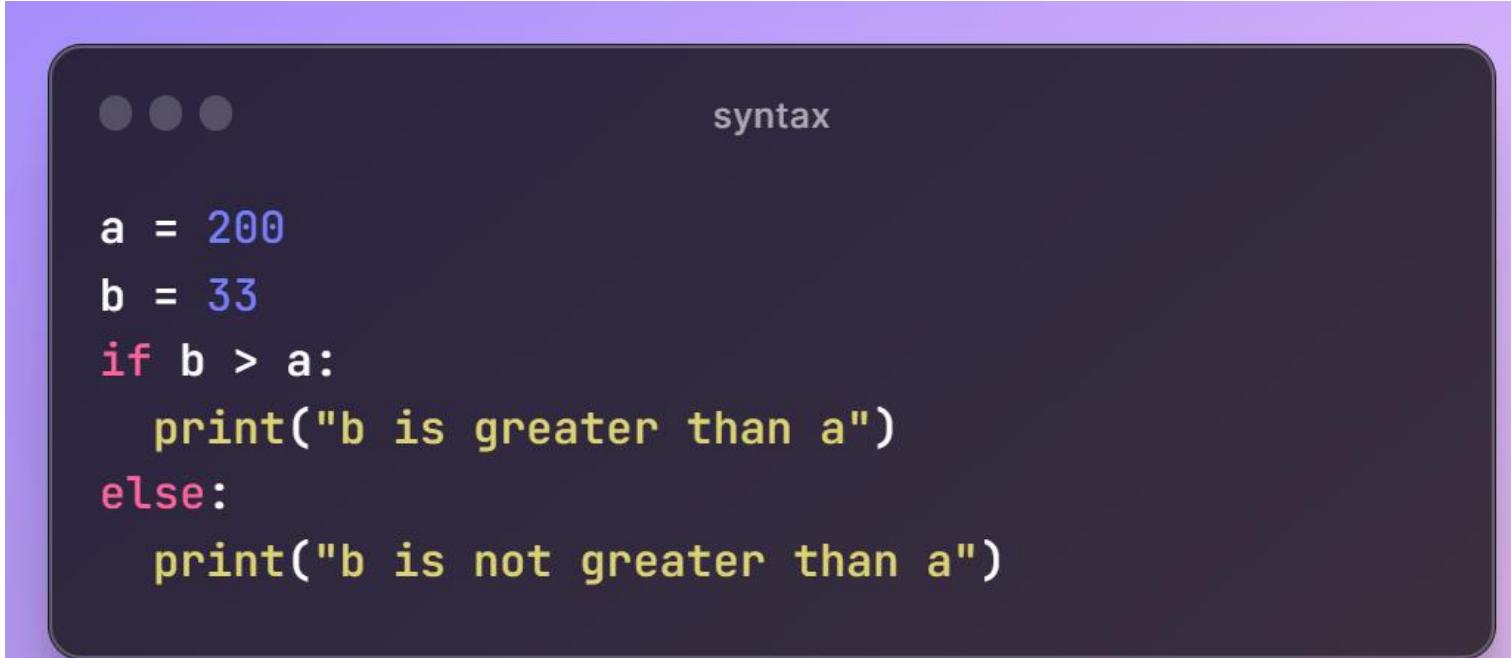
syntax

```
• • •  
a = 200  
b = 33  
if b > a:  
    print("b is greater than a")  
elif a == b:  
    print("a and b are equal")  
else:  
    print("a is greater than b")
```

PYTHON

Else

You can also have an **else** without the **elif**:



The image shows a dark-themed code editor window with a light purple header bar. In the top right corner of the header bar, the word "syntax" is written in white. The main area of the code editor contains the following Python code:

```
...  
a = 200  
b = 33  
if b > a:  
    print("b is greater than a")  
else:  
    print("b is not greater than a")
```

PYTHON

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.



syntax

One line if statement:

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

PYTHON

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:



The image shows a dark-themed code editor window with a light purple header bar. In the top right corner of the header bar, there are three small circular icons. To the right of these icons, the word "syntax" is written in a light gray font. The main area of the code editor is dark gray with white text. At the top left of this area, there are three small white dots. Below these dots, the text "One line if else statement:" is displayed in white. Underneath this heading, there is some Python code. The variable "a" is assigned the value 2, and the variable "b" is assigned the value 330. A single-line if-else statement is shown, where "print("A")" is executed if "a" is greater than "b", and "print("B")" is executed otherwise. The code is color-coded: "print" and the strings "A" and "B" are in yellow, while the comparison operator ">" is in green, and the "if" and "else" keywords are in pink.

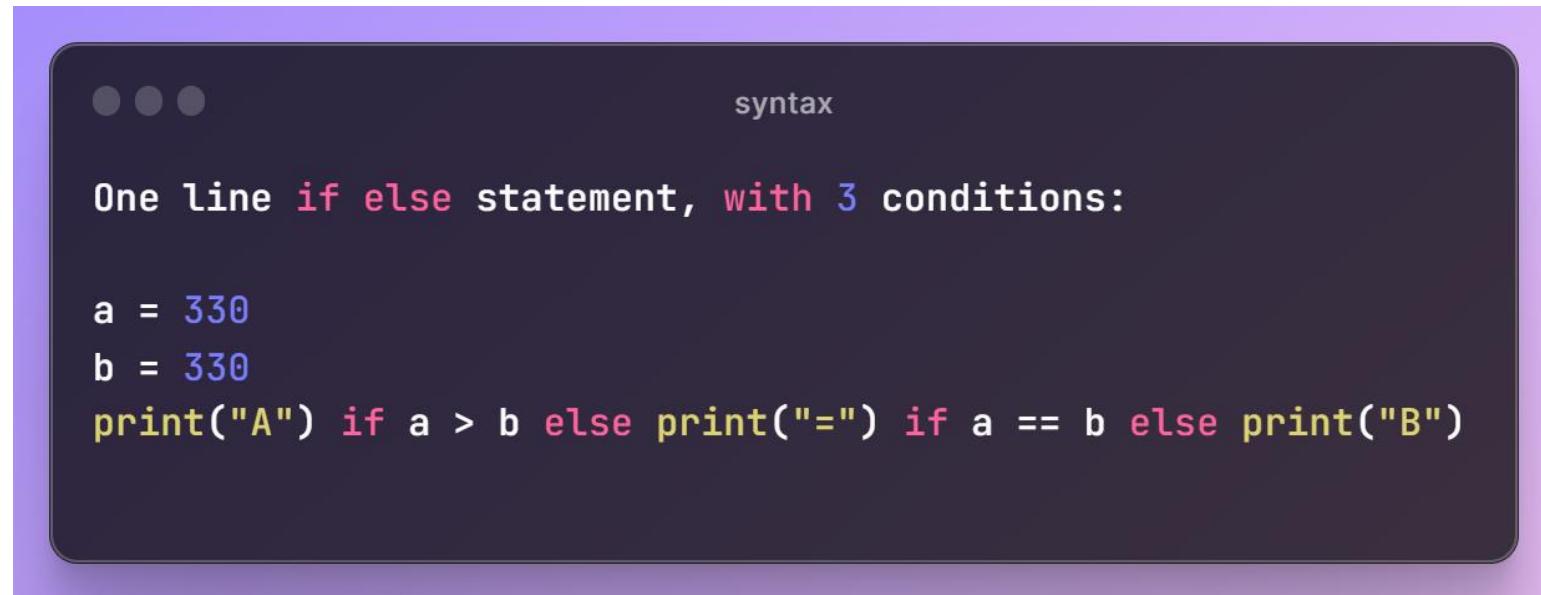
```
a = 2  
b = 330  
print("A") if a > b else print("B")
```

PYTHON

Short Hand If

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:



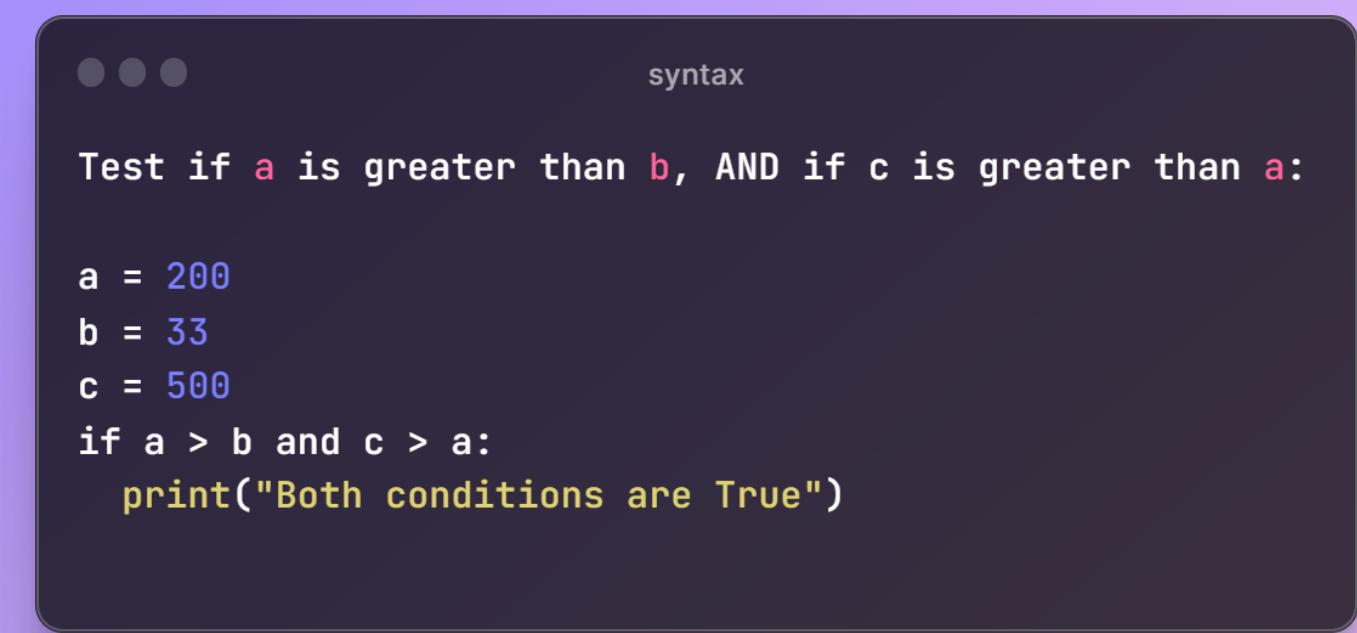
The image shows a terminal window with a dark background and light-colored text. At the top left, there are three grey dots. To the right of the terminal area, the word "syntax" is written in a smaller font. Inside the terminal, the text reads:

```
One line if else statement, with 3 conditions:  
  
a = 330  
b = 330  
print("A") if a > b else print( "=" ) if a == b else print("B")
```

PYTHON

And

The **and** keyword is a logical operator, and is used to combine conditional statements:



The screenshot shows a dark-themed code editor window with a light purple header bar. In the top right corner of the header bar, there are three small gray dots. To the right of these dots, the word "syntax" is written in white. Below the header, the code is displayed in white text on a dark background. The code is as follows:

```
... syntax

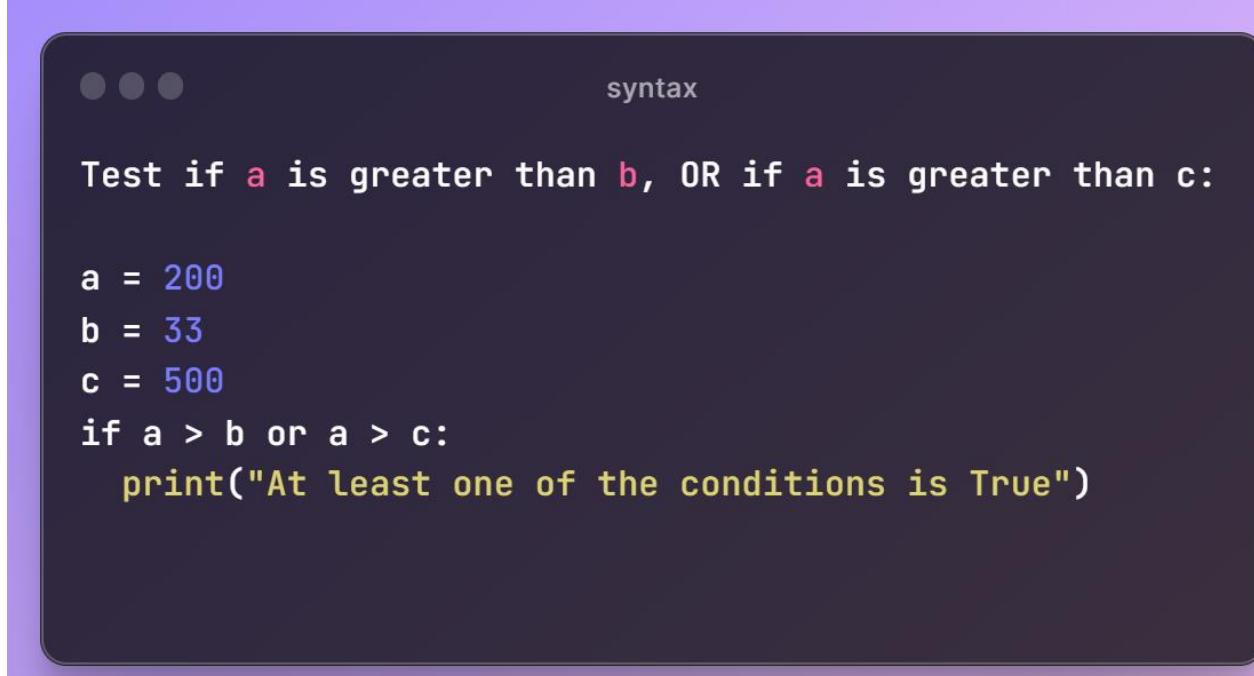
Test if a is greater than b, AND if c is greater than a:

a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

PYTHON

Or

The **or** keyword is a logical operator, and is used to combine conditional statements:



```
...syntax

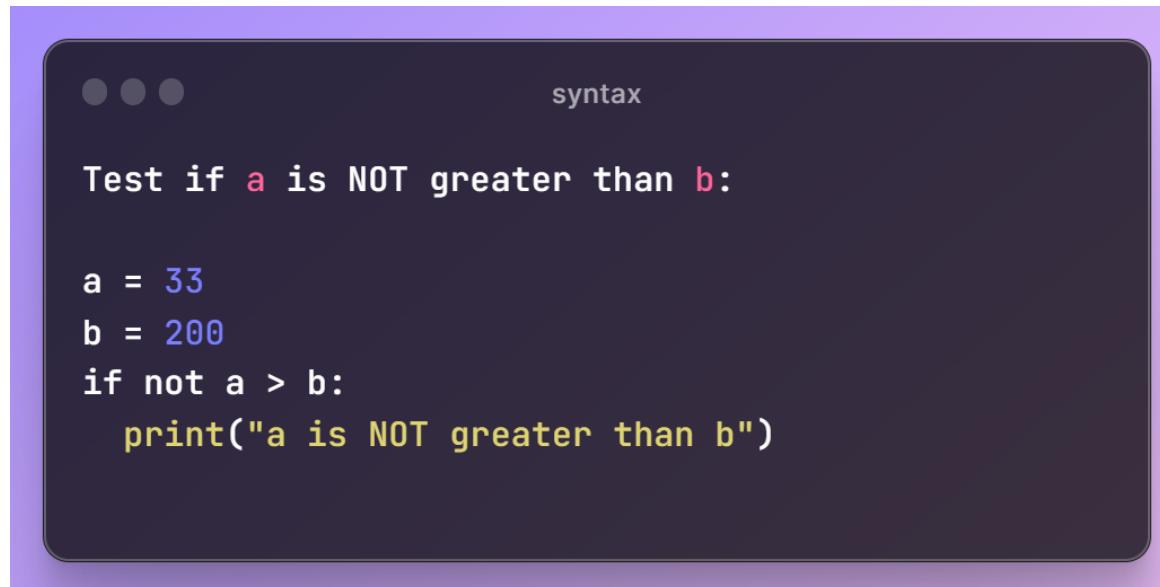
Test if a is greater than b, OR if a is greater than c:

a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

PYTHON

Not

The **not** keyword is a logical operator, and is used to reverse the result of the conditional statement:



The code editor window shows the following Python code:

```
● ● ●           syntax

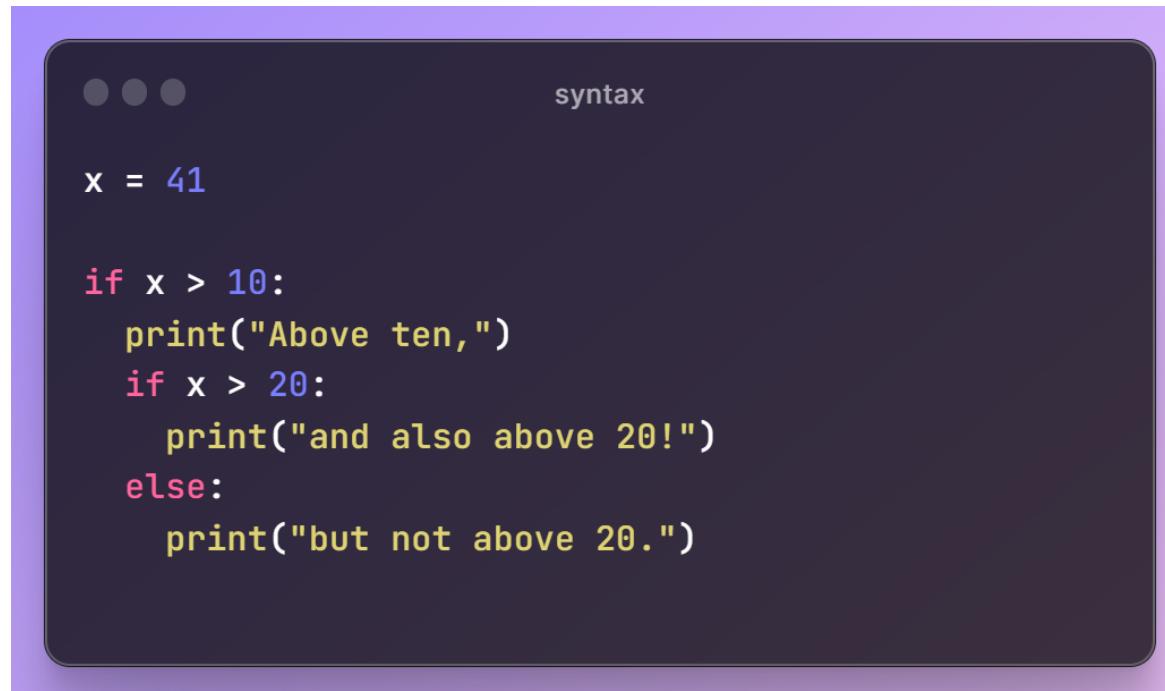
Test if a is NOT greater than b:

a = 33
b = 200
if not a > b:
    print("a is NOT greater than b")
```

PYTHON

Nested If

You can have **if** statements inside **if** statements, this is called *nested if* statements.



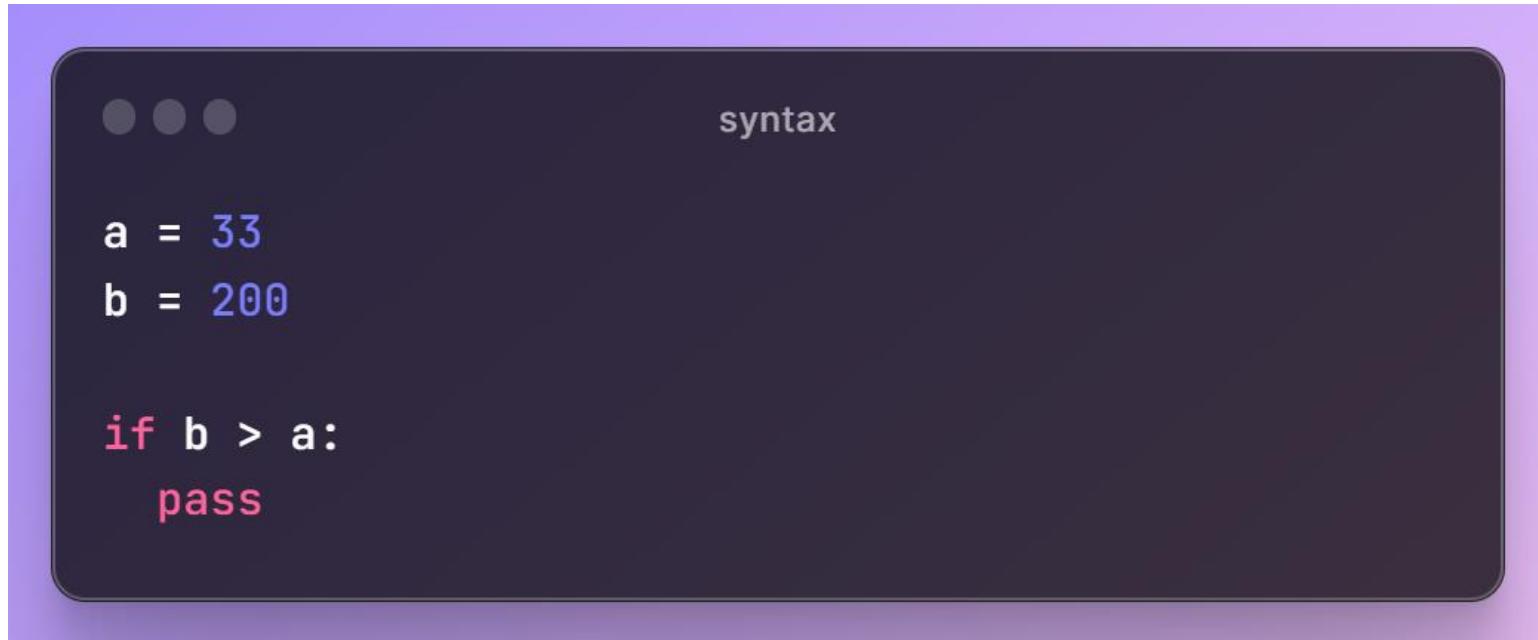
The image shows a dark-themed code editor window with a purple header bar. In the top right corner of the header bar, there are three small circular icons. To the right of these icons, the word "syntax" is written in a light color. The main area of the code editor contains the following Python code:

```
x = 41

if x > 10:
    print("Above ten,")
    if x > 20:
        print("and also above 20!")
    else:
        print("but not above 20.")
```

The Pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.



A screenshot of a dark-themed code editor window. In the top right corner, the word "syntax" is displayed. On the left side of the editor, there are three gray circular icons. The code itself is as follows:

```
...  
a = 33  
b = 200  
  
if b > a:  
    pass
```

PYTHON

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.

Note: remember to increment i, or else the loop will continue forever.

The **while** loop requires relevant variables to be ready, in this example we need to define an indexing variable, **i**, which we set to 1.



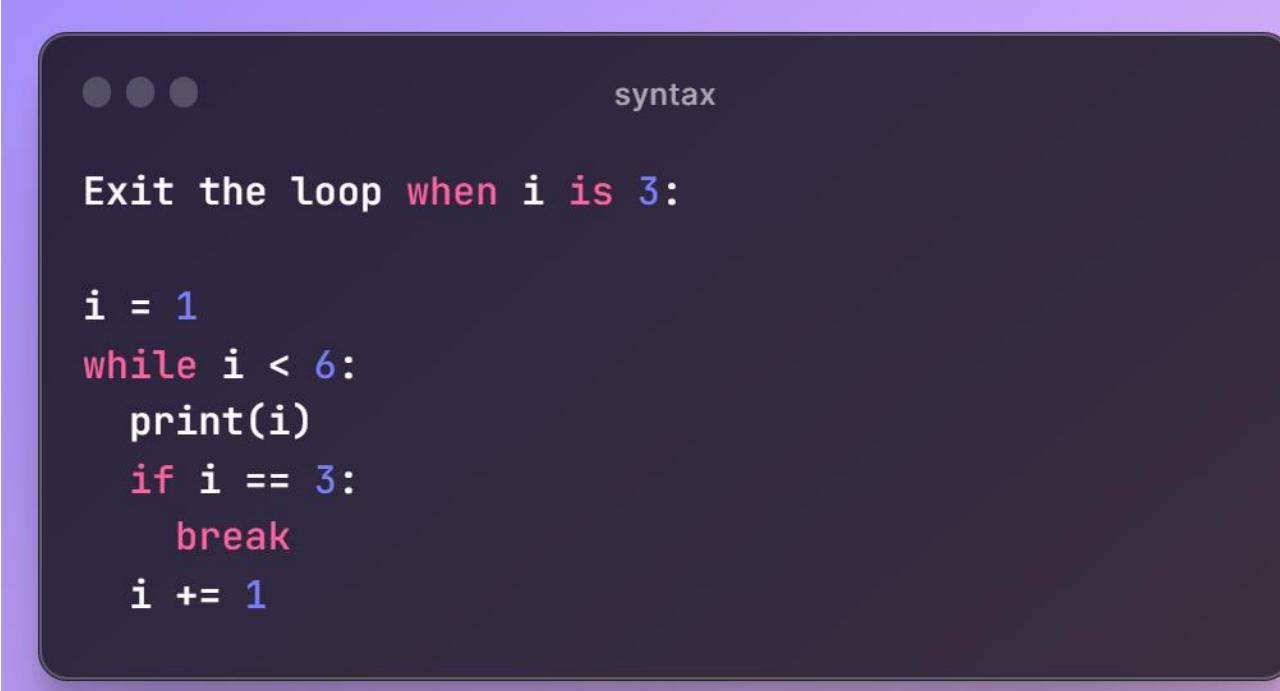
syntax

Print i as long as i is less than 6:

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

The Break Statement

With the **break** statement we can stop the loop even if the while condition is true:



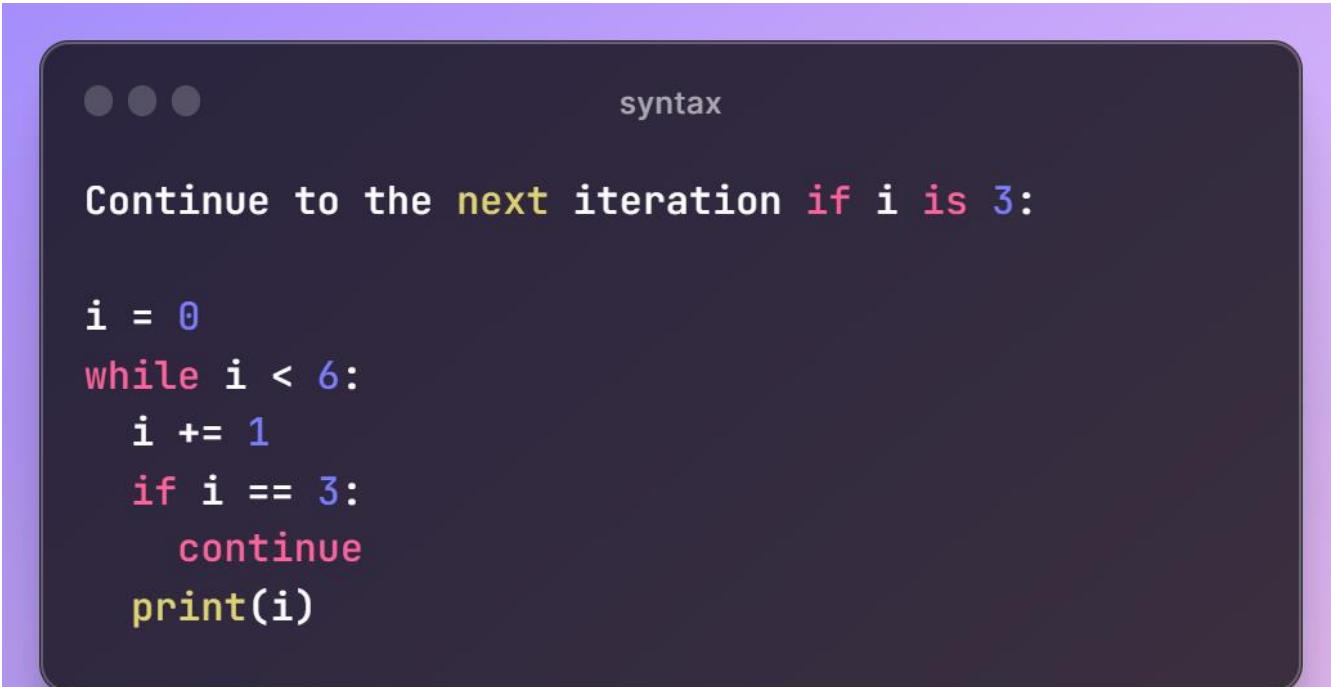
```
... syntax

Exit the loop when i is 3:

i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

The Continue Statement

With the **continue** statement we can stop the current iteration, and continue with the next:



syntax

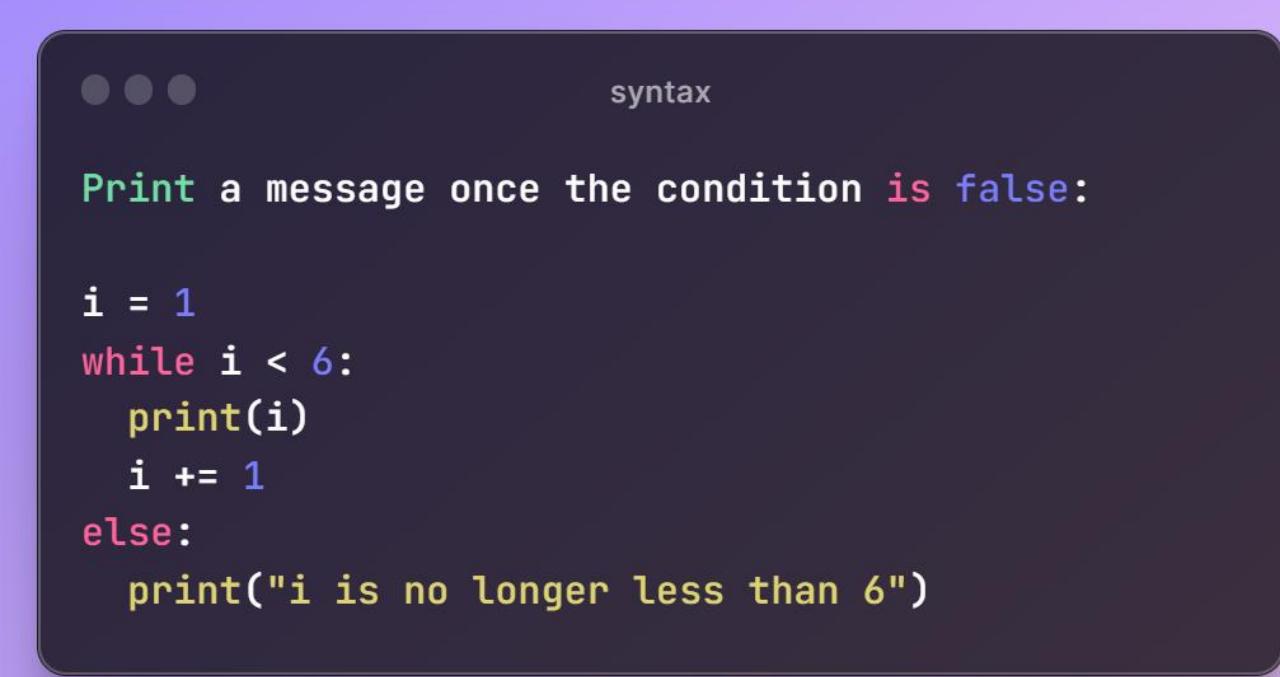
```
Continue to the next iteration if i is 3:

i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

PYTHON

The Else Statement

With the **else** statement we can run a block of code once when the condition no longer is true:



The image shows a dark-themed mobile phone screen with three dots at the top left. The word "syntax" is at the top right. The main area contains the following Python code:

```
Print a message once the condition is false:  
  
i = 1  
while i < 6:  
    print(i)  
    i += 1  
else:  
    print("i is no longer less than 6")
```

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.
- The **for** loop does not require an indexing variable to set beforehand.



... syntax

Print each fruit in a fruit list:

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

PYTHON

Looping through a String

Even strings are iterable objects, they contain a sequence of characters:



syntax

Loop through the letters in the word "banana":

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

Break Statement

With the **break** statement we can stop the loop before it has looped through all the items:



syntax

```
Exit the loop when x is "banana":  
  
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

PYTHON

Break Statement

• • •

syntax

Exit the loop `when x is "banana"`, but `this time the break comes before the print:`

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

The Continue Statement

With the **continue** statement we can stop the current iteration of the loop, and continue with the next:



Do not print banana:

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

The `range()` Function

- To loop through a set of code a specified number of times, we can use the `range()` function,
- The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

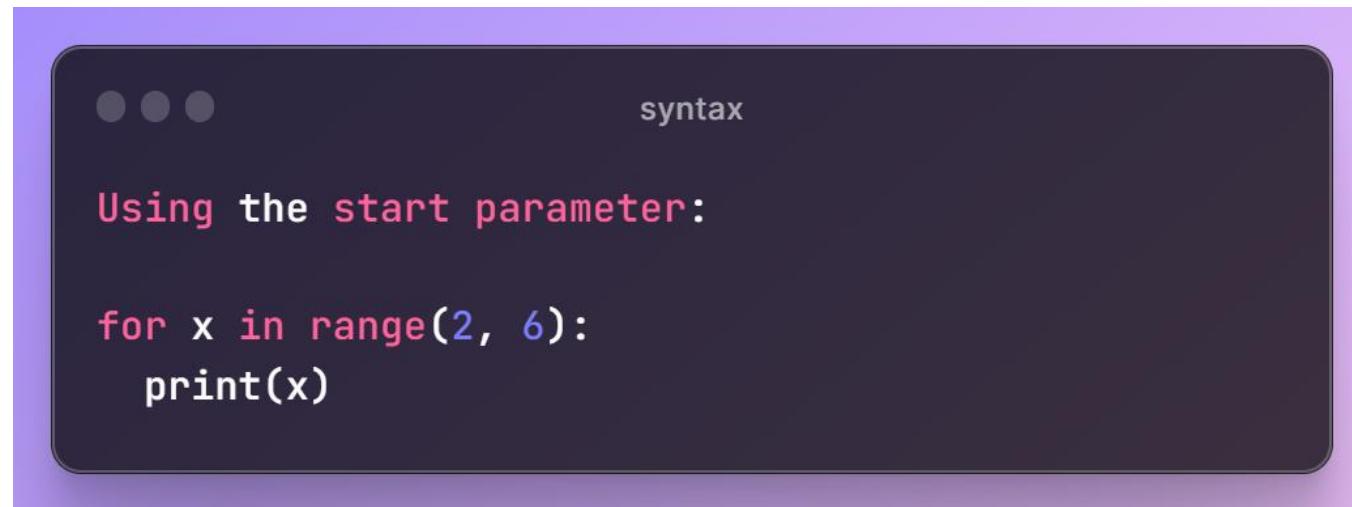
- Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.



PYTHON

The `range()` Function

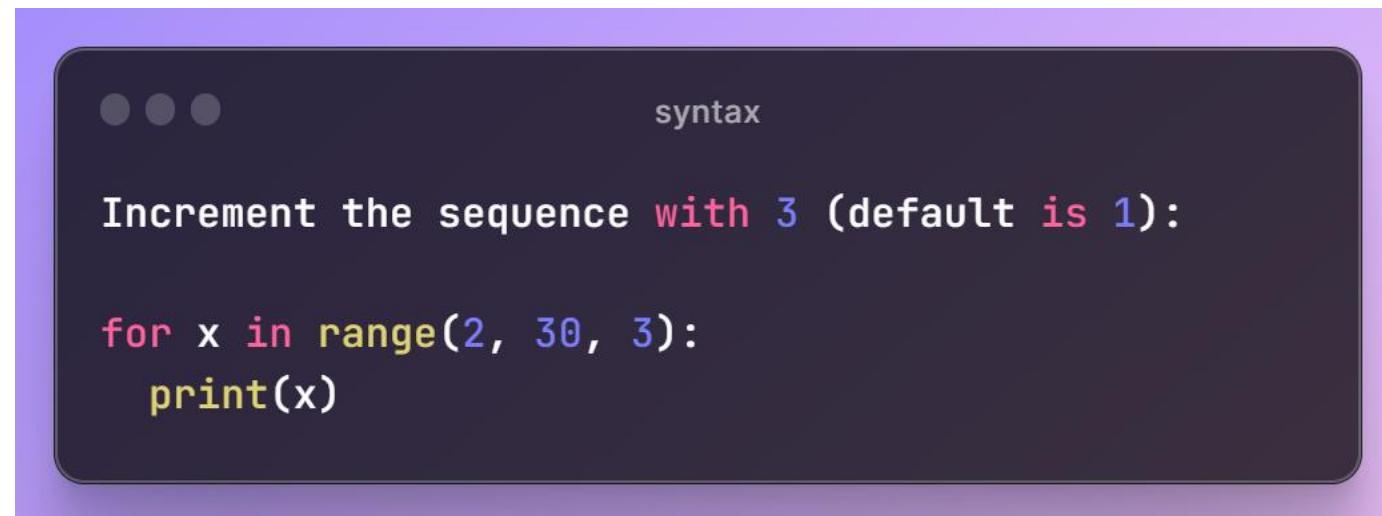
- The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):



PYTHON

The range() Function

- The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:



PYTHON

Else in For Loop

The **else** keyword in a **for** loop specifies a block of code to be executed when the loop is finished:

Note: The **else** block will NOT be executed if the loop is stopped by a **break** statement.

The image shows a screenshot of a Python code editor with two code snippets. Both snippets are enclosed in dark grey boxes with three dots at the top left and the word 'syntax' at the top right.

Left Snippet:

```
Print all numbers from 0 to 5, and print a message when the loop has ended:  
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

Right Snippet:

```
Break the loop when x is 3, and see what happens with the else block:  
for x in range(6):  
    if x == 3: break  
    print(x)  
else:  
    print("Finally finished!")
```

Nested Loop

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":



The image shows a dark-themed code editor window with a purple header bar. In the top right corner of the header bar, there are three small white dots and the word "syntax". The main area of the window contains Python code. The code starts with a comment "Print each adjective for every fruit:", followed by two lists of words: "adj = ["red", "big", "tasty"]" and "fruits = ["apple", "banana", "cherry"]". Below these lists is a nested loop structure: "for x in adj:" followed by "for y in fruits:". Underneath the inner loop, the word "print(x, y)" is shown. The entire code block is highlighted with a light blue color.

```
••• syntax

Print each adjective for every fruit:

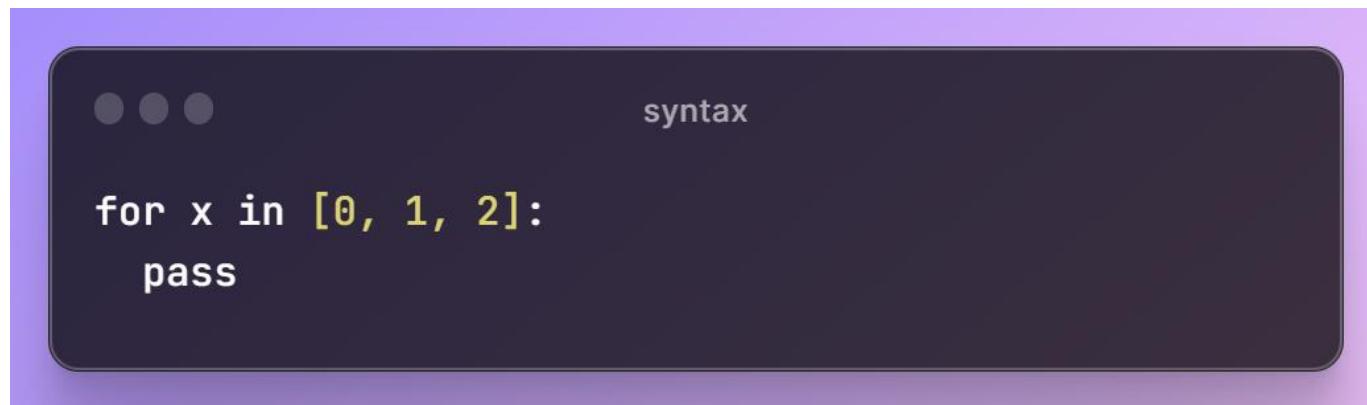
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

PYTHON

The pass Statement

- **for** loops cannot be empty, but if you for some reason have a **for** loop with no content, put in the **pass** statement to avoid getting an error.



PYTHON

Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Creating a Function

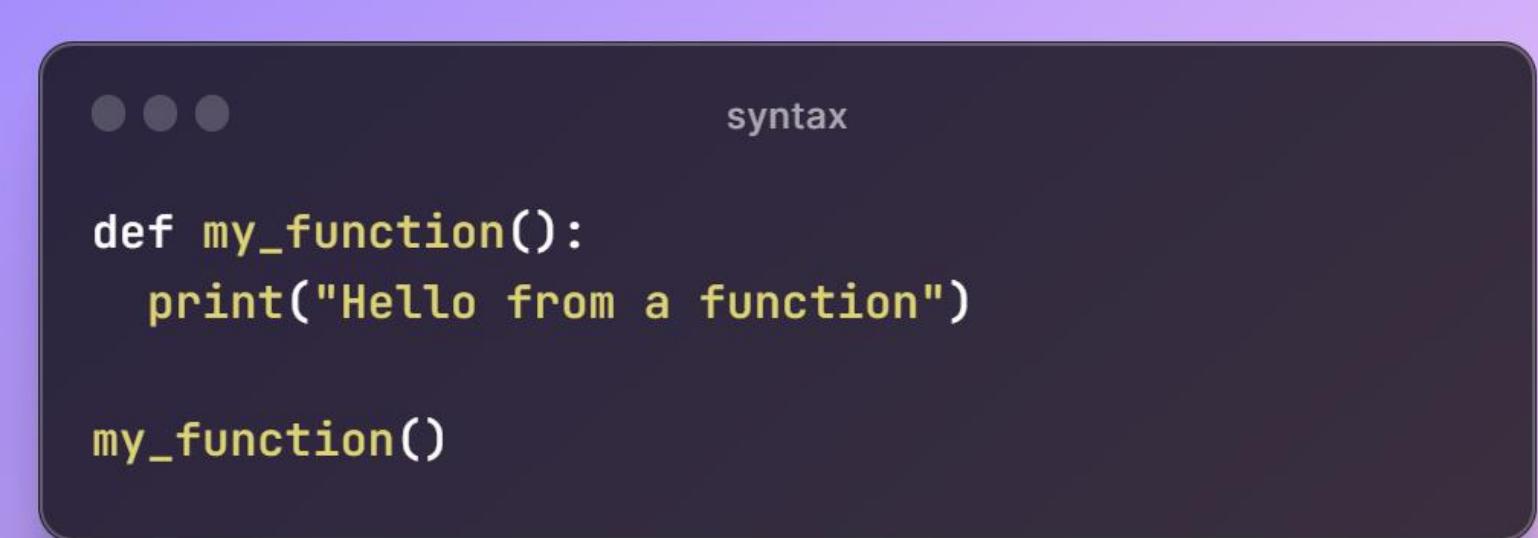
- In Python a function is defined using the **def** keyword:



PYTHON

Calling a Functions

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



```
... syntax

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

my_function()
```

Arguments

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
- The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

PYTHON

Arguments

Arguments are often shortened to *args* in Python documentations.

```
def my_function(fname):
    print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Parameters Or Arguments ?

- The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.
- From a function's perspective:
 - A parameter is the variable listed inside the parentheses in the function definition.
 - An argument is the value that is sent to the function when it is called.

Number of Arguments

- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.
- Example

• • •

syntax

This **function** expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
    print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

PYTHON

Number of Arguments

If you try to call the function with 1 or 3 arguments, you will get an error:



The screenshot shows a terminal window with a dark background and light-colored text. At the top left, there are three small circular icons. To the right of them, the word "syntax" is displayed. Below this, a message is shown in green text: "This function expects 2 arguments, but gets only 1:". Underneath the message, a Python script is displayed in green text. The script defines a function named "my_function" that takes two arguments, "fname" and "lname", and prints them together with a space between them. Then, it calls the function with a single argument, "Emil".

```
... syntax  
This function expects 2 arguments, but gets only 1:  
  
def my_function(fname, lname):  
    print(fname + " " + lname)  
  
my_function("Emil")
```

Arbitrary Arguments, *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a *tuple* of arguments, and can access the items accordingly:

Arbitrary Arguments are often shortened to ***args** in Python documentations.

Example

- If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])  
  
my_function("Emil", "Tobias", "Linus")
```

Keyword Arguments

- You can also send arguments with the *key = value* syntax.
- This way the order of the arguments does not matter.

The phrase ***Keyword Arguments*** are often shortened to ***kwargs*** in Python documentations.

Example

```
...  
  
def my_function(child3, child2, child1):  
    print("The youngest child is " + child3)  
  
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Arbitrary Keyword Arguments, **kwargs

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk: `**` before the parameter name in the function definition.
- This way the function will receive a *dictionary* of arguments, and can access the items accordingly :

Example

- If the number of keyword arguments is unknown, add a double `**` before the parameter name:

```
def my_function(**kid):
    print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value:

```
...  
  
def my_function(country = "Norway"):  
    print("I am from " + country)  
  
my_function("Sweden")  
my_function("India")  
my_function()  
my_function("Brazil")
```

Passing a List as an Argument

- You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.
- E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example

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

PYTHON

Return Values

- To let a function return a value, use the **return** statement:

Example

```
...  
def my_function(x):  
    return 5 * x  
  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

The pass Statement

- **function** definitions cannot be empty, but if you for some reason have a **function** definition with no content, put in the **pass** statement to avoid getting an error.

Example

```
...  
def myfunction():  
    pass
```

Positional-Only Arguments

- You can specify that a function can have ONLY positional arguments, or ONLY keyword arguments.
- To specify that a function can have only positional arguments, add `, /` after the arguments:

Example

```
def my_function(x, /):
    print(x)

my_function(3)
```

Positional-Only Arguments

- Without the `,` / you are actually allowed to use keyword arguments even if the function expects positional arguments:

Example

```
...  
def my_function(x):  
    print(x)  
  
my_function(x = 3)
```

Positional-Only Arguments

- But when adding the `,` / you will get an error if you try to send a keyword argument:

Example

```
def my_function(x, /):
    print(x)

my_function(x = 3)
```

Keyword-Only Arguments

- To specify that a function can have only keyword arguments, add `*`, *before* the arguments:

Example

```
def my_function(*, x):
    print(x)

my_function(x = 3)
```

Keyword-Only Arguments

- Without the *, you are allowed to use positional arguments even if the function expects keyword arguments:

Example

```
def my_function(x):  
    print(x)  
  
my_function(3)
```

Keyword-Only Arguments

- But when adding the *, / you will get an error if you try to send a positional argument:

Example

```
def my_function(*, x):
    print(x)

my_function(3)
```

PYTHON

Combine Positional-Only and Keyword-Only

- You can combine the two argument types in the same function.
- Any argument *before* the `/`, are positional-only, and any argument *after* the `*`, are keyword-only.

```
...  
def my_function(a, b, /, *, c, d):  
    print(a + b + c + d)  
  
my_function(5, 6, c = 7, d = 8)
```

Recursion

- Python also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements `(-1)` every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).
- To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

PYTHON

Recursion

Recursion Example

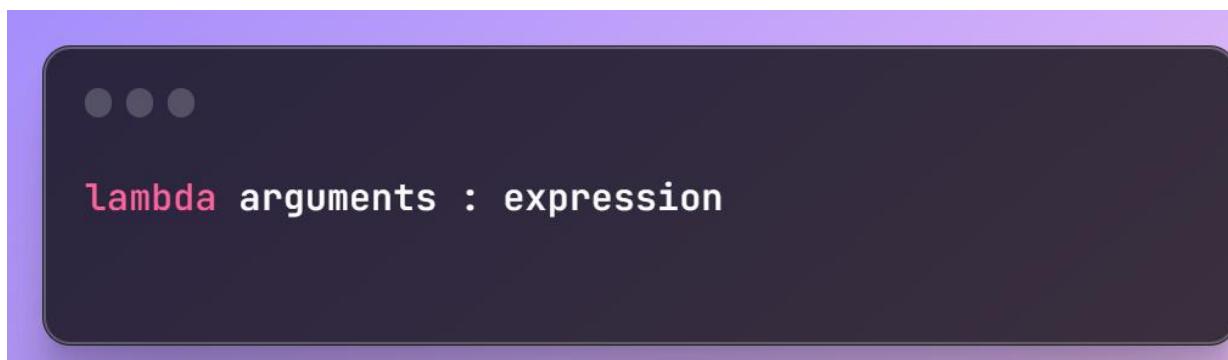
```
...  
  
def tri_recursion(k):  
    if(k > 0):  
        result = k + tri_recursion(k - 1)  
        print(result)  
    else:  
        result = 0  
    return result  
  
print("\n\nRecursion Example Results")  
tri_recursion(6)
```

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

Example

- Add 10 to argument a, and return the result:

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

Lambda

- Lambda functions can take any number of arguments:

Example

- Multiply argument **a** with argument **b** and return the result:

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

PYTHON

Lambda

Example

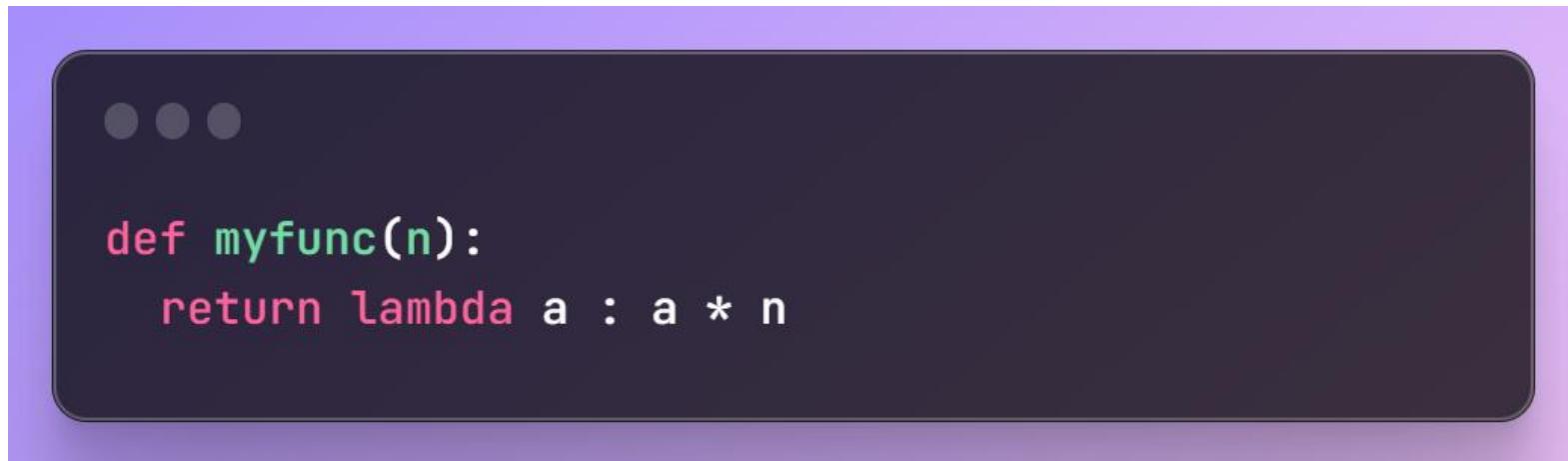
- Summarize argument a, b, and c and return the result:

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

Lambda

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
```

PYTHON

Lambda

- Use that function definition to make a function that always doubles the number you send in:

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

mydoubler = myfunc(2)

print(mydoubler(11))
```

PYTHON

Lambda

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

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

PYTHON

Lambda

- Or, use the same function definition to make both functions, in the same program:

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

PYTHON

Classes and Objects

Python Classes/Objects

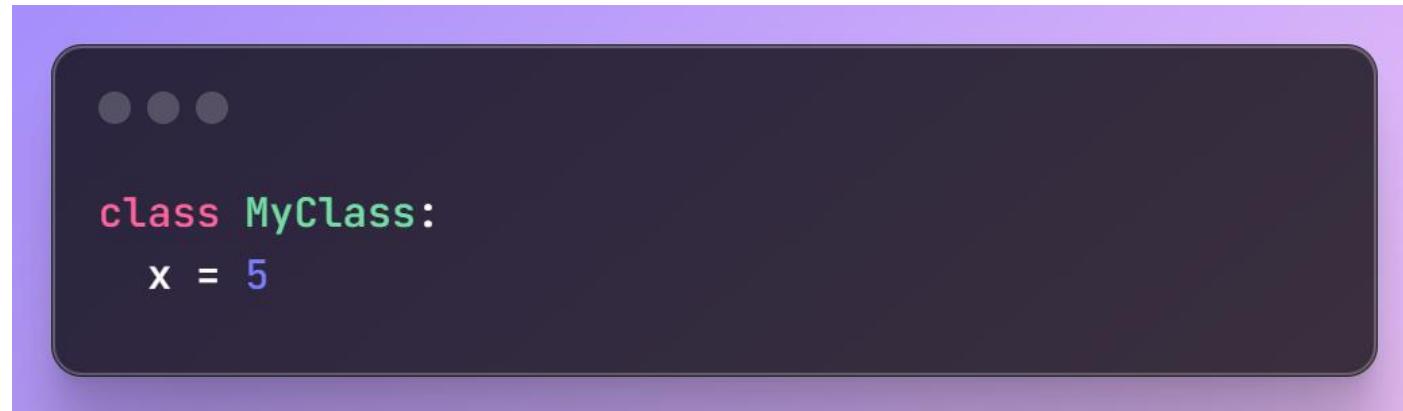
- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

PYTHON

Classes and Objects

Create a Class

- To create a class, use the keyword **class**:



A screenshot of a dark-themed code editor window. At the top left, there are three small circular icons. Below them, the code is displayed in white text on a black background. The code defines a class named 'MyClass' with a single attribute 'x' set to the value 5.

```
class MyClass:  
    x = 5
```

Classes and Objects

Create a Object

- Now we can use the class named MyClass to create objects:

Example:

- Create an object named p1, and print the value of x:

```
...  
p1 = MyClass()  
print(p1.x)
```

Classes and Objects

The `__init__()` Function:

- The examples above are classes and objects in their simplest form, and are not really useful in real life applications.
- To understand the meaning of classes we have to understand the built-in `__init__()` function.
- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Classes and Objects

Example:

- Create a class named Person, use the `__init__()` function to assign values for name and age:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
p1 = Person("John", 36)  
  
print(p1.name)  
print(p1.age)
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Classes and Objects

The `__str__()` Function:

- The `__str__()` function controls what should be returned when the class object is represented as a string.
- It is basically a string representation of the class.
- If the `__str__()` function is not set, the string representation of the object is returned:

Classes and Objects

Example:

- The string representation of an object WITHOUT the `__str__()` function:

The image shows a dark-themed code editor window with a purple header bar. The code in the editor is as follows:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
p1 = Person("John", 36)  
  
print(p1)
```

The code defines a `Person` class with an `__init__` method to initialize `name` and `age`. It then creates an instance `p1` and prints it.

Classes and Objects

Example:

- The string representation of an object WITH the `__str__()` function:

```
• • •  
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def __str__(self):  
        return f"{self.name}({self.age})"  
  
p1 = Person("John", 36)  
  
print(p1)
```

Classes and Objects

Object Methods:

- Objects can also contain methods. Methods in objects are functions that belong to the object.
- Let us create a method in the Person class:

Classes and Objects

Example:

- Insert a function that prints a greeting, and execute it on the p1 object:

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

```
• • •

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Classes and Objects

The **self** Parameter:

- The **self** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named **self** , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Classes and Objects

Example:

- Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:  
    def __init__(mysillyobject, name, age):  
        mysillyobject.name = name  
        mysillyobject.age = age  
  
    def myfunc(abc):  
        print("Hello my name is " + abc.name)  
  
p1 = Person("John", 36)  
p1.myfunc()
```

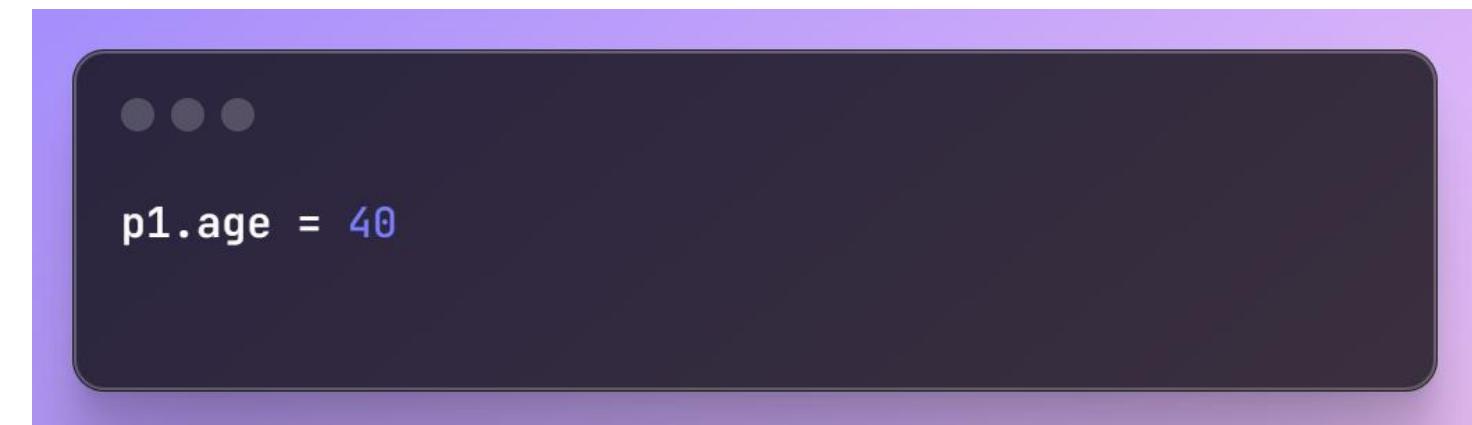
Classes and Objects

Modify Object Properties:

- You can modify properties on objects like this:

Example:

Set the age of p1 to 40:



Classes and Objects

Delete Object Properties:

- You can delete properties on objects by using the del keyword:

Example:

Delete the age property from the p1 object:

```
del p1.age
```

Classes and Objects

Delete Objects:

- You can delete objects by using the del keyword:

Example:

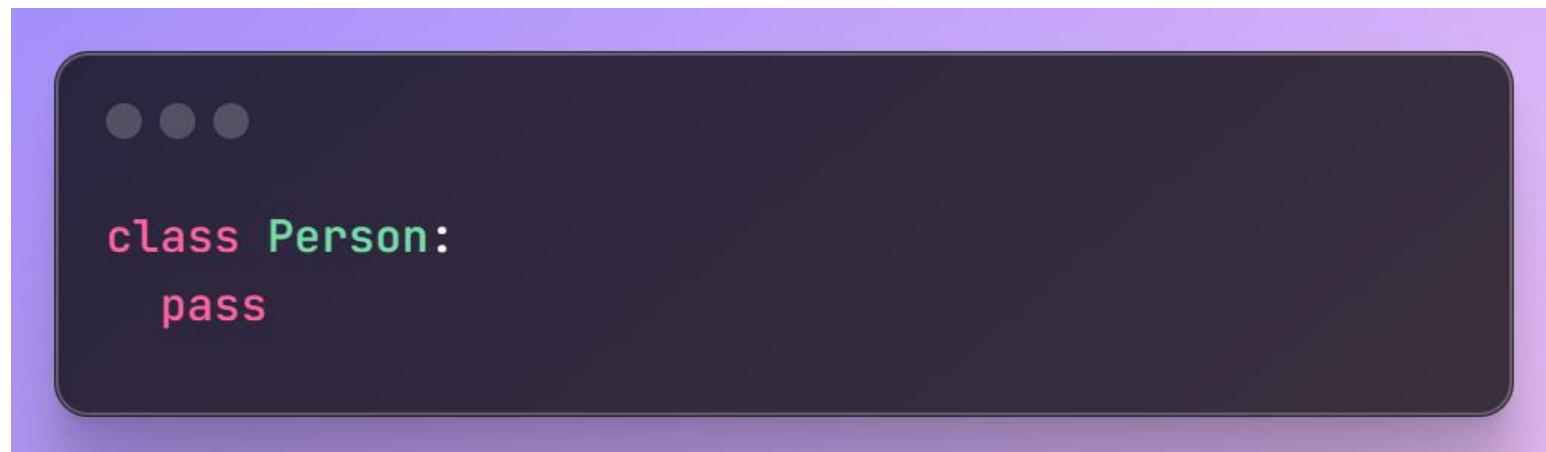
Delete the p1 object:

```
del p1
```

Classes and Objects

The pass Statement:

- `class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the `pass` statement to avoid getting an error.



A screenshot of a code editor window showing a Python code snippet. The code defines a class named `Person` with a single `pass` statement. The code is displayed in a dark-themed editor with a light purple background. The class definition is preceded by three gray ellipsis dots.

```
...  
class Person:  
    pass
```

Inheritance

Python Inheritance:

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

PYTHON

Inheritance

Create a Parent Class:

- Create a class named Person, with **firstname** and **lastname** properties, and a **printname** method:

```
class Person:  
    def __init__(self, fname, lname):  
        self.firstname = fname  
        self.lastname = lname  
  
    def printname(self):  
        print(self.firstname, self.lastname)  
  
#Use the Person class to create an object,  
#and then execute the printname method:  
  
x = Person("John", "Doe")  
x.printname()
```

Inheritance

Create a Child Class:

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:
- Create a class named **Student**, which will inherit the properties and methods from the **Person** class:
- Now the Student class has the same properties and methods as the Person class.

```
...  
class Student(Person):  
    pass
```

Note: Use the **pass** keyword when you do not want to add any other properties or methods to the class.

PYTHON

Inheritance

Example:

- Use the **Student** class to create an object, and then execute the printname method:

```
...  
x = Student("Mike", "Olsen")  
x.printname()
```

Inheritance

Add the `__init__()` Function:

- So far we have created a child class that inherits the properties and methods from its parent.
- We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Inheritance

Example:

- Add the `__init__()` function to the `Student` class:

```
...  
  
class Student(Person):  
    def __init__(self, fname, lname):  
        #add properties etc.
```

- When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Inheritance

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

- To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

- Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Inheritance

Use the `super()` Function:

- Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):  
    def __init__(self, fname, lname):  
        super().__init__(fname, lname)
```

- By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

PYTHON

Inheritance

Add Properties: Add a property called **graduationyear** to the **Student** class:

Example

- Add a property called **graduationyear** to the **Student** class:

```
class Student(Person):  
    def __init__(self, fname, lname):  
        super().__init__(fname, lname)  
        self.graduationyear = 2019
```

Inheritance

- In the example below, the year **2019** should be a variable, and passed into the **Student** class when creating student objects. To do so, add another parameter in the **__init__()** function:

Example

- Add a **year** parameter, and pass the correct year when creating objects:

```
...  
class Student(Person):  
    def __init__(self, fname, lname, year):  
        super().__init__(fname, lname)  
        self.graduationyear = year  
  
x = Student("Mike", "Olsen", 2019)
```

Inheritance

Add Methods: Example: Add a method called `welcome` to the `Student` class:

```
...
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year

    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

- If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Iterators

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterators

Iterator vs Iterable:

- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.
- All these objects have a `iter()` method which is used to get an iterator:

Iterators

Example

- Return an iterator from a tuple, and print each value:

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

print(next(myit))
print(next(myit))
print(next(myit))
```

Iterators

- Even strings are iterable objects, and can return an iterator:

Example: Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Iterators

Looping Through an Iterator: We can also use a for loop to iterate through an iterable object:

Example:

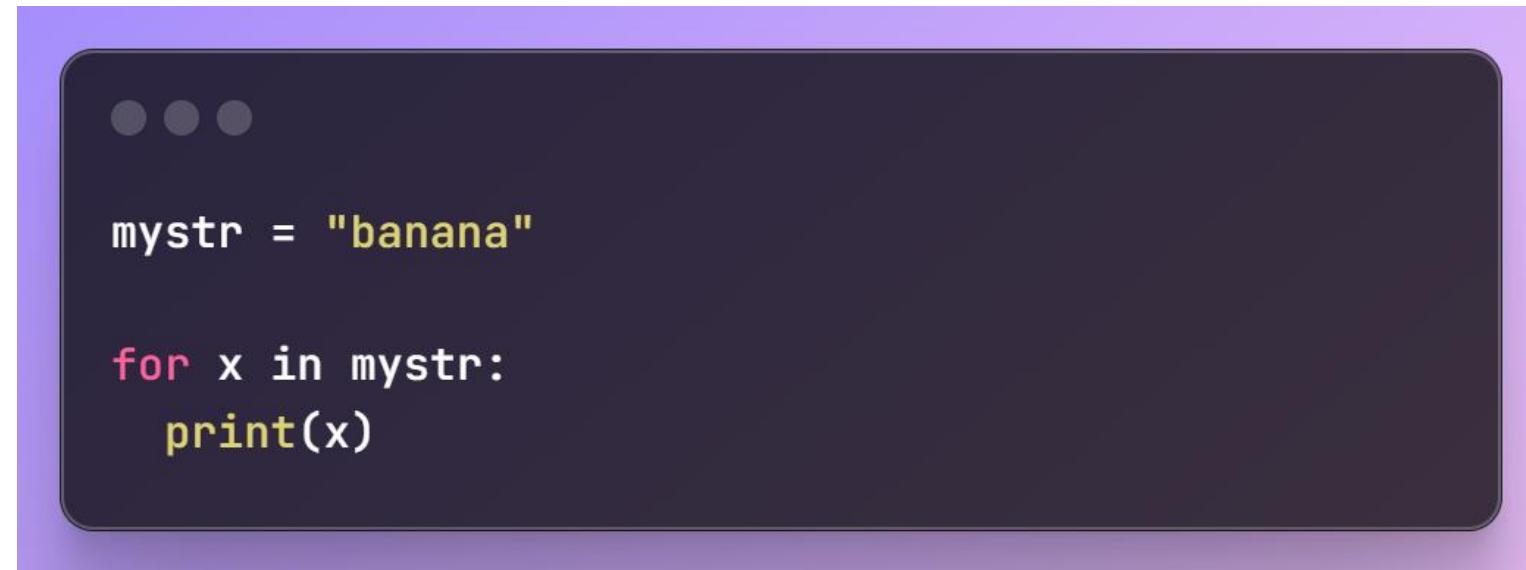
- Iterate the values of a tuple:

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

Iterators

Example:

- Iterate the characters of a string:



```
...  
  
mystr = "banana"  
  
for x in mystr:  
    print(x)
```

- The **for** loop actually creates an iterator object and executes the **next()** method for each loop.

Iterators

Create an Iterator:

- To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.
- As you have learned in the [Python Classes/Objects](#) chapter, all classes have a function called `__init__()`, which allows you to do some initializing when the object is being created.
- The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
- The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

Iterators

Example: Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
...
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        x = self.a
        self.a += 1
        return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Iterators

StopIteration

- The example above would continue forever if you had enough next() statements, or if it was used in a **for** loop.
- To prevent the iteration from going on forever, we can use the **StopIteration** statement.
- In the **__next__()** method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

PYTHON

Iterators

Example: Stop after 20 iterations:

```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 20:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

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

Polymorphism

- The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Function Polymorphism

- An example of a Python function that can be used on different objects is the `len()` function.

Polymorphism

String: For strings, `len()` returns the number of characters:

```
...  
x = "Hello World!"  
  
print(len(x))
```

Polymorphism

Tuple: For tuples, `len()` returns the number of items in it:

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

Polymorphism

Dictionary: For dictionaries, `len()` returns the number of key/value pairs in it:

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

Polymorphism

Class Polymorphism:

- Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.
- For example, say we have three classes: **Car**, **Boat**, and **Plane**, and they all have a method called **move()**:

Polymorphism

Example:

- Different classes with the same method:
- Look at the for loop at the end. Because of polymorphism we can execute the same method for all three classes.

```
class Car:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def move(self):  
        print("Drive!")  
  
class Boat:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def move(self):  
        print("Sail!")  
  
class Plane:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def move(self):  
        print("Fly!")  
  
car1 = Car("Ford", "Mustang")      #Create a Car class  
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class  
plane1 = Plane("Boeing", "747")    #Create a Plane class  
  
for x in (car1, boat1, plane1):  
    x.move()
```

Polymorphism

Inheritance Class Polymorphism:

- What about classes with child classes with the same name? Can we use polymorphism there?
- Yes. If we use the example above and make a parent class called **Vehicle**, and make **Car**, **Boat**, **Plane** child classes of **Vehicle**, the child classes inherits the Vehicle methods, but can override them:

Polymorphism

Example

- Create a class called **Vehicle** and make **Car, Boat, Plane** child classes of **Vehicle**:

```
...
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def move(self):
        print("Move!")

class Car(Vehicle):
    pass

class Boat(Vehicle):
    def move(self):
        print("Sail!")

class Plane(Vehicle):
    def move(self):
        print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
    print(x.brand)
    print(x.model)
    x.move()
```

Polymorphism

- Child classes inherits the properties and methods from the parent class.
- In the example above you can see that the **Car** class is empty, but it inherits **brand**, **model**, and **move()** from **Vehicle**.
- The **Boat** and **Plane** classes also inherit **brand**, **model**, and **move()** from **Vehicle**, but they both override the **move()** method.
- Because of polymorphism we can execute the same method for all classes.

PYTHON

Scope

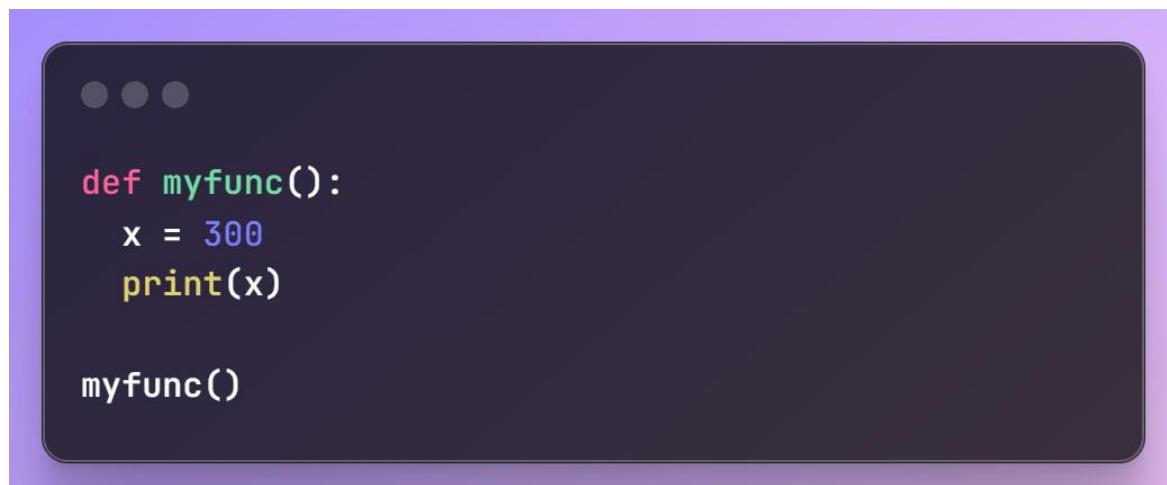
- A variable is only available from inside the region it is created. This is called **scope**.

Local Scope:

- A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Example:

- A variable created inside a function is available inside that function:



```
def myfunc():
    x = 300
    print(x)

myfunc()
```

Scope

Function Inside Function

- As explained in the example above, the variable `x` is not available outside the function, but it is available for any function inside the function:

Example:

- The local variable can be accessed from a function within the function:

```
...  
def myfunc():  
    x = 300  
    def myinnerfunc():  
        print(x)  
    myinnerfunc()  
  
myfunc()
```

PYTHON

Scope

Global Scope

- A variable created in the main body of the Python code is a global variable and belongs to the global scope.
- Global variables are available from within any scope, global and local.

Example

- A variable created outside of a function is global and can be used by anyone:

```
...
x = 300

def myfunc():
    print(x)

myfunc()

print(x)
```

PYTHON

Scope

Naming Variables

- If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example

- The function will print the local **x**, and then the code will print the global **x**:

```
...
x = 300

def myfunc():
    x = 200
    print(x)

myfunc()

print(x)
```

Scope

Global Keyword

- If you need to create a **global** variable, but are stuck in the local scope, you can use the **global** keyword.
- The **global** keyword makes the variable global.

Example

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

```
...  
def myfunc():  
    global x  
    x = 300  
  
myfunc()  
  
print(x)
```

PYTHON

Scope

- Also, use the **global** keyword if you want to make a change to 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 = 300

def myfunc():
    global x
    x = 200

myfunc()

print(x)
```

PYTHON

Modules

What is a Module?

- Consider a module to be the same as a code library.
- A file containing a set of functions you want to include in your application.

Create a Module: To create a module just save the code you want in a file with the file extension **.py**:

Example: Save this
code in a file
named **mymodule.py**

```
def greeting(name):  
    print("Hello, " + name)
```

PYTHON

Modules

Use a Module

- Now we can use the module we just created, by using the **import** statement:

Example

- Import the module named **mymodule**, and call the greeting function:

```
...  
import mymodule  
  
mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax: *module_name.function_name*.

Modules

Variables in Module:

- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

- Save this code in the file **mymodule.py**

```
person1 = {  
    "name": "John",  
    "age": 36,  
    "country": "Norway"  
}
```

PYTHON

Modules

Example

- Import the module named mymodule, and access the person1 dictionary:



```
import mymodule

a = mymodule.person1["age"]
print(a)
```

PYTHON

Modules

Naming a Module:

- You can name the module file whatever you like, but it must have the file extension **.py**

Re-naming a Module:

- You can create an alias when you import a module, by using the **as** keyword:

Example

- Create an alias
for **mymodule** called **mx**:

```
...  
import mymodule as mx  
  
a = mx.person1["age"]  
print(a)
```

PYTHON

Modules

Built-in Modules:

- There are several built-in modules in Python, which you can import whenever you like.

Example:

- Import and use the **platform** module:

```
import platform

x = platform.system()
print(x)
```

Modules

Using the `dir()` Function:

- There is a built-in function to list all the function names (or variable names) in a module.
The `dir()` function:

Example:

- List all the defined names belonging to the platform module:

```
...  
  
import platform  
  
x = dir(platform)  
print(x)
```

Note: The `dir()` function can be used on *all* modules, also the ones you create yourself.

Modules

Import From Module:

- You can choose to import only parts from a module, by using the `from` keyword.

Example:

- The module named `mymodule` has one function and one dictionary:

```
...  
def greeting(name):  
    print("Hello, " + name)  
  
person1 = {  
    "name": "John",  
    "age": 36,  
    "country": "Norway"  
}
```

Modules

Example:

- Import only the person1 dictionary from the module:

```
from mymodule import person1  
  
print (person1["age"])
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module. Example: `person1["age"]`, **not** `mymodule.person1["age"]` .

PYTHON

Dates

Python Dates:

- A date in Python is not a data type of its own, but we can import a module named **datetime** to work with dates as date objects.

Example

- Import the datetime module and display the current date:

```
...  
import datetime  
  
x = datetime.datetime.now()  
print(x)
```

Dates

Date Output:

- When we execute the code from the example above the result will be:
2024-03-08 19:14:51.808624
- The date contains year, month, day, hour, minute, second, and microsecond.
- The **datetime** module has many methods to return information about the date object.
- Here are a few examples, you will learn more about them later in this chapter:

Dates

Example

- Return the year and name of weekday:

```
import datetime  
  
x = datetime.datetime.now()  
  
print(x.year)  
print(x.strftime("%A"))
```

Dates

Creating Date Objects:

- To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.
- The `datetime()` class requires three parameters to create a date: year, month, day.

Example

- Create a date object:

```
import datetime  
  
x = datetime.datetime(2020, 5, 17)  
  
print(x)
```

The `datetime()` class also takes parameters for time and timezone (hour, minute, second, microsecond, `tzone`), but they are optional, and has a default value of `0`, (`None` for timezone).

Dates

The strftime() Method:

- The `datetime` object has a method for formatting date objects into readable strings.
- The method is called `strftime()`, and takes one parameter, `format`, to specify the `format` of the returned string:

Example

- Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

Math

- Python has a set of built-in math functions, including an extensive math module, that allows you to perform mathematical tasks on numbers.

Built-in Math Functions

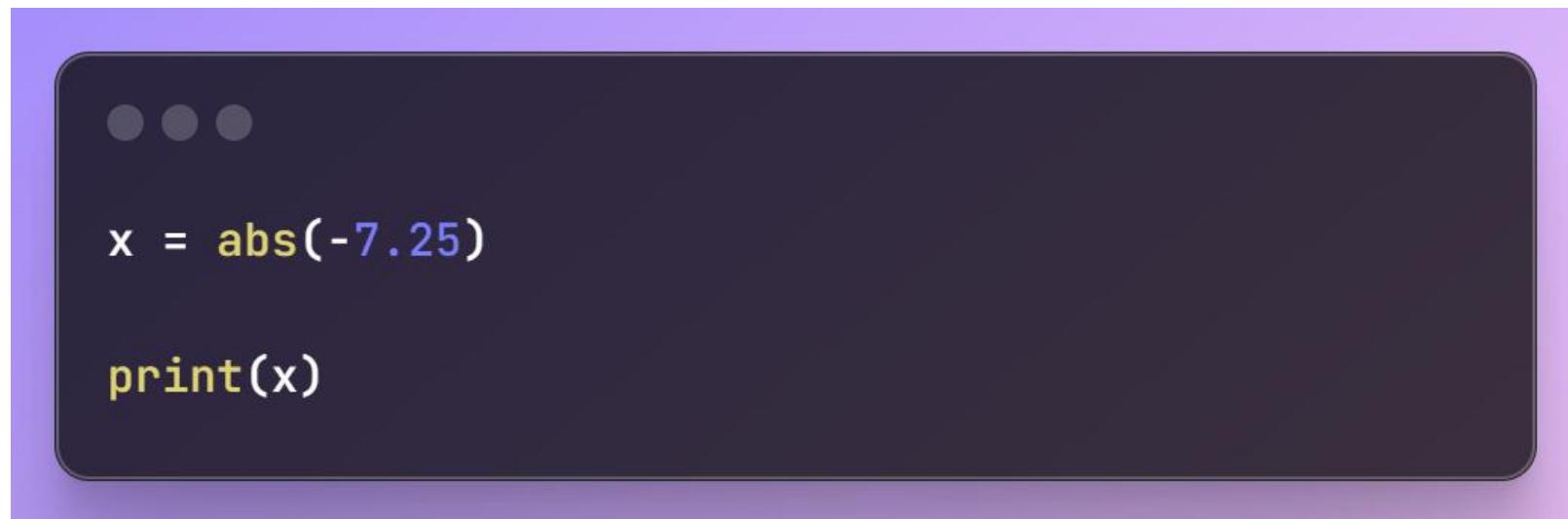
- The `min()` and `max()` functions can be used to find the lowest or highest value in an iterable:

```
...  
x = min(5, 10, 25)  
y = max(5, 10, 25)  
  
print(x)  
print(y)
```

PYTHON

Math

- The **abs()** function returns the absolute (positive) value of the specified number:



```
...  
x = abs(-7.25)  
  
print(x)
```

PYTHON

Math

- The `pow(x, y)` function returns the value of x to the power of y (x^y).
- Return the value of 4 to the power of 3 (same as $4 * 4 * 4$):

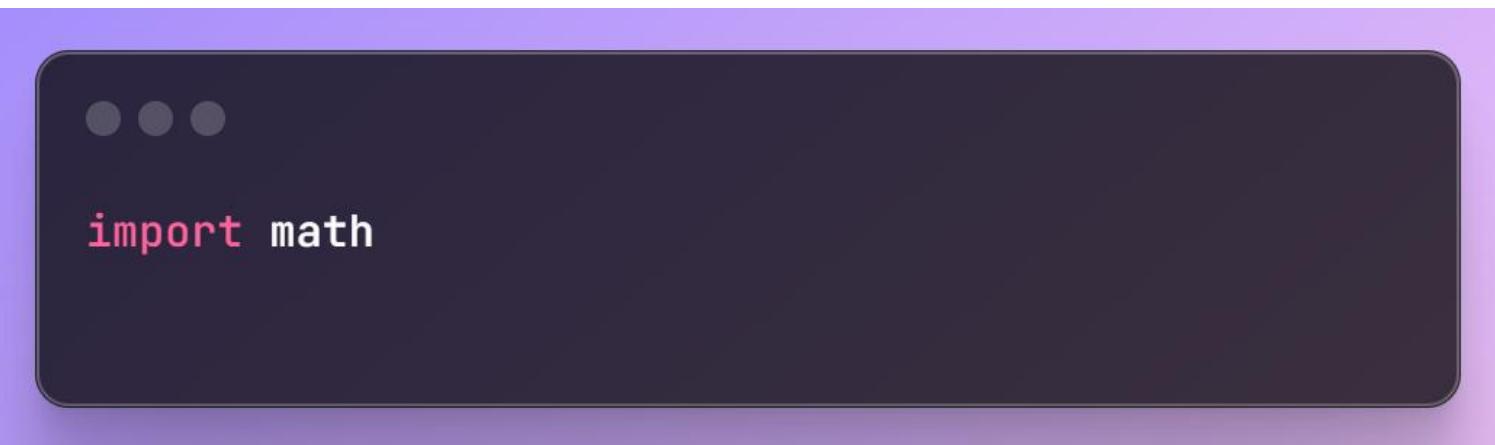
```
...  
x = pow(4, 3)  
  
print(x)
```

PYTHON

Math

The Math Module:

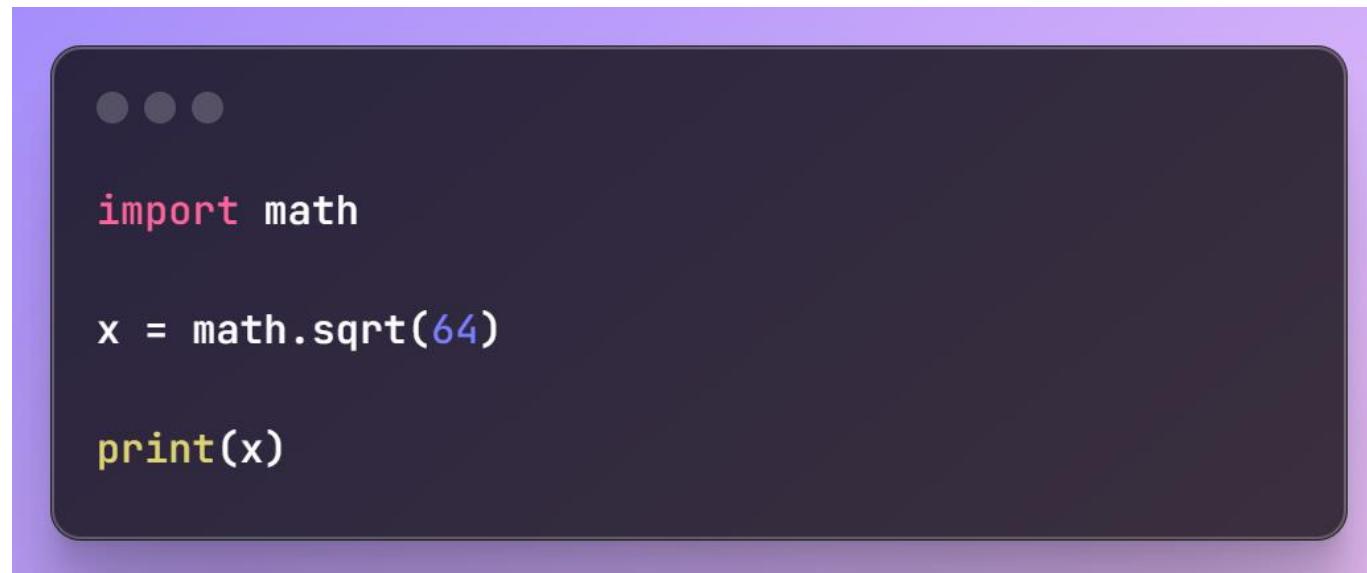
- Python has also a built-in module called **math**, which extends the list of mathematical functions.
- To use it, you must import the **math** module:



PYTHON

Math

- When you have imported the `math` module, you can start using methods and constants of the module.
- The `math.sqrt()` method for example, returns the square root of a number:



```
import math

x = math.sqrt(64)

print(x)
```

PYTHON

Math

- The `math.ceil()` method rounds a number upwards to its nearest integer, and the `math.floor()` method rounds a number downwards to its nearest integer, and returns the result:

```
import math

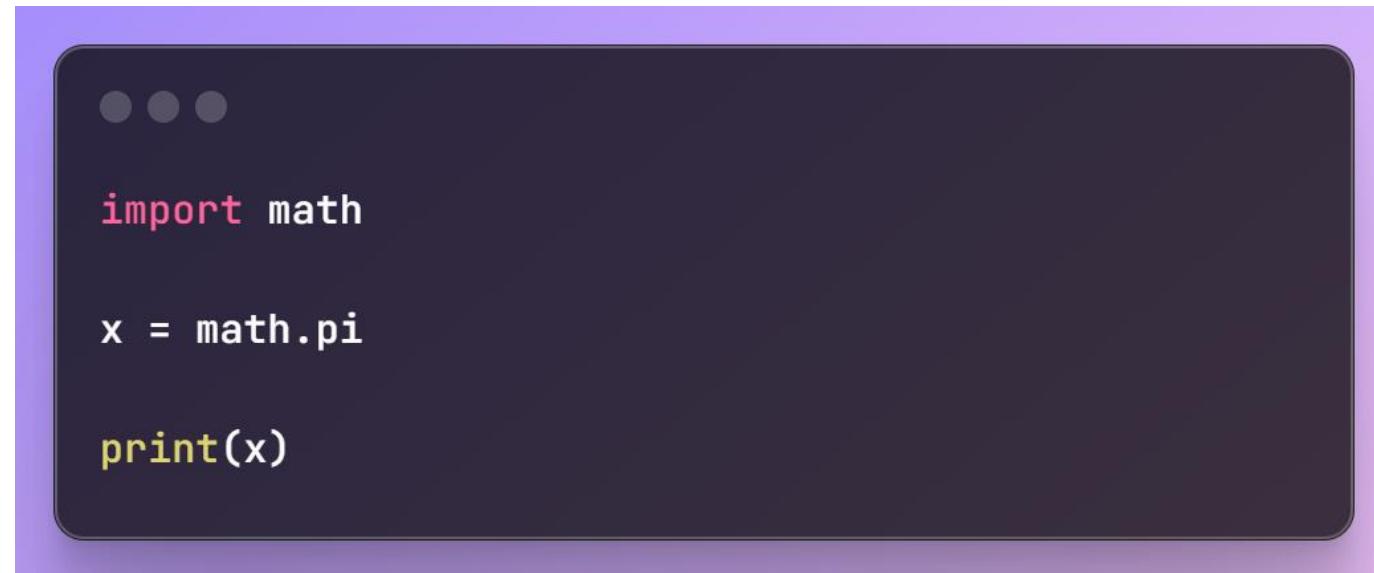
x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1
```

PYTHON

Math

- The `math.pi` constant, returns the value of PI (3.14...):



```
import math

x = math.pi

print(x)
```

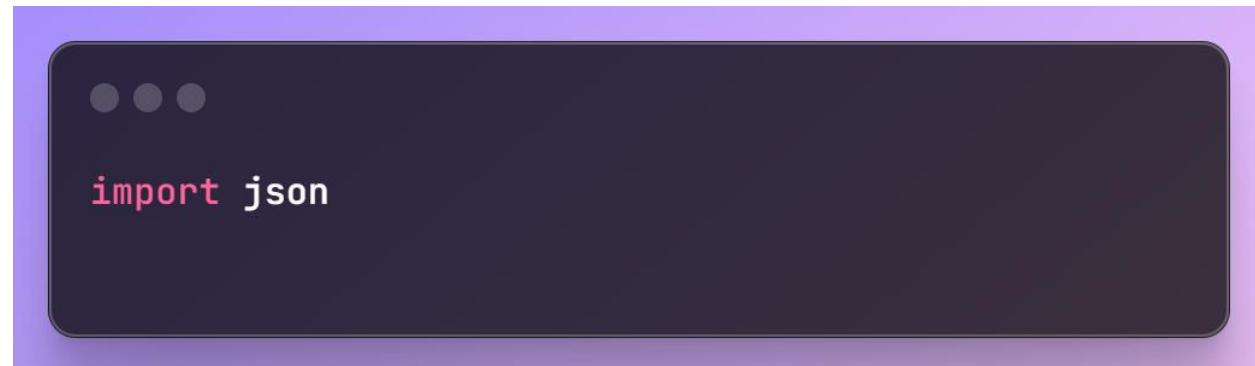
PYTHON

JSON

- JSON is a syntax for storing and exchanging data.
- JSON is text, written with JavaScript object notation.
- Python has a built-in package called **json**, which can be used to work with JSON data.

Example

- Import the **json** module:



```
...  
import json
```

PYTHON

JSON

Parse JSON - Convert from JSON to Python:

- If you have a JSON string, you can parse it by using the `json.loads()` method.
- The result will be a Python dictionary.

PYTHON

JSON

Example: Convert from JSON to Python:

```
import json

# some JSON:
x = '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

PYTHON

JSON

Convert from Python to JSON:

- If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method.

Example:

- Convert from Python to JSON:

```
import json

# a Python object (dict):
x = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

PYTHON

JSON

- You can convert Python objects of the following types, into JSON strings:
 - dict
 - list
 - tuple
 - string
 - int
 - float
 - True
 - False
 - None

PYTHON

JSON

Example:

- Convert Python objects into JSON strings, and print the values:

```
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

PYTHON

JSON

- When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

PYTHON

JSON

Example

- Convert a Python object containing all the legal data types:

```
import json

x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann","Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}

print(json.dumps(x))
```

JSON

Format the Result

- The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.
- The `json.dumps()` method has parameters to make it easier to read the result:

Example

- Use the `indent` parameter to define the numbers of indents:

```
...  
json.dumps(x, indent=4)
```

PYTHON

JSON

- You can also define the separators, default value is (" ", ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

Example

- Use the **separators** parameter to change the default separator:

```
json.dumps(x, indent=4, separators=(". ", " = "))
```

JSON

Order the Result

- The `json.dumps()` method has parameters to order the keys in the result:

Example

- Use the `sort_keys` parameter to specify if the result should be sorted or not:

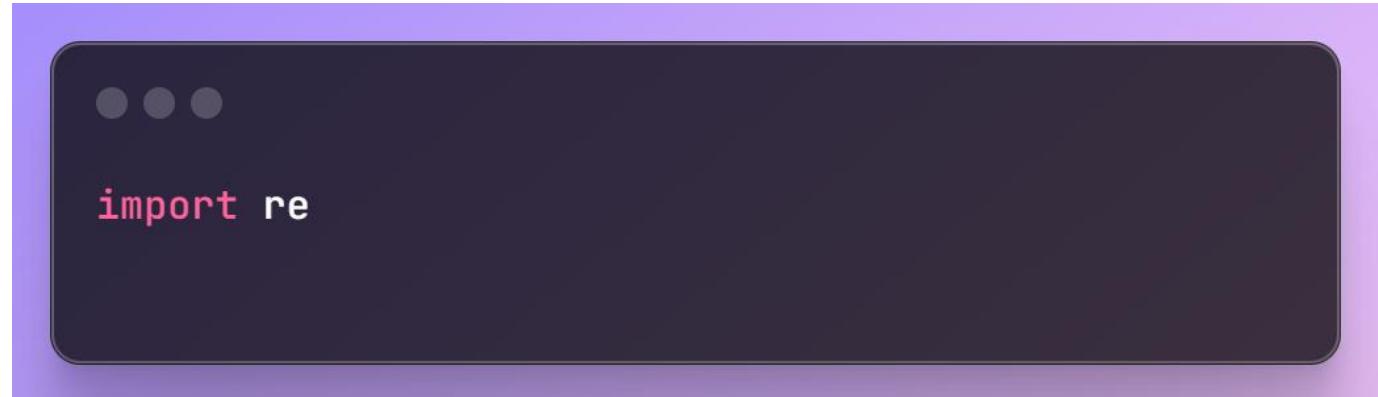
```
...  
json.dumps(x, indent=4, sort_keys=True)
```

RegEx

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.

RegEx Module

- Python has a built-in package called **re**, which can be used to work with Regular Expressions.
- Import the **re** module:



RegEx

RegEx in Python

- When you have imported the `re` module, you can start using regular expressions:

Example

- Search the string to see if it starts with "The" and ends with "Spain":

```
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

RegEx

RegEx Functions

- The `re` module offers a set of functions that allows us to search a string for a match:

Function	Description
<code>.findall</code>	Returns a list containing all matches
<code>search</code>	Returns a <code>Match object</code> if there is a match anywhere in the string
<code>split</code>	Returns a list where the string has been split at each match
<code>sub</code>	Replaces one or many matches with a string

RegEx

The `findall()` Function:

- The `findall()` function returns a list containing all matches.

Example:

- Print a list of all matches:
- The list contains the matches in the order they are found.
- If no matches are found, an empty list is returned:

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

PYTHON

RegEx

Example

- Return an empty list if no match was found:

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

RegEx

The search() Function:

- The `search()` function searches the string for a match, and returns a Match object if there is a match.
- If there is more than one match, only the first occurrence of the match will be returned:

RegEx

Example:

- Search for the first white-space character in the string:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())
```

PYTHON

RegEx

- If no matches are found, the value **None** is returned:

```
import re

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)
```

RegEx

The `split()` Function:

- The `split()` function returns a list where the string has been split at each match:

Example:

- Split at each white-space character:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

RegEx

- You can control the number of occurrences by specifying the **maxsplit** parameter:

Example

- Split the string only at the first occurrence:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt, 1)
print(x)
```

RegEx

The `sub()` Function:

- The `sub()` function replaces the matches with the text of your choice:

Example:

- Replace every white-space character with the number 9:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

RegEx

- You can control the number of replacements by specifying the **count** parameter:

Example

- Replace the first 2 occurrences:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
```

RegEx

Match Object:

- A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value **None** will be returned, instead of the Match Object.

PYTHON

RegEx

Example:

- Do a search that will return a Match Object:

```
import re

txt = "The rain in Spain"
x = re.search("ai", txt)
print(x) #this will print an object
```

RegEx

- The Match object has properties and methods used to retrieve information about the search, and the result:

`.span()` returns a tuple containing the start-, and end positions of the match.

`.string` returns the string passed into the function.

`.group()` returns the part of the string where there was a match.

PYTHON

RegEx

Example:

- Print the position (start- and end-position) of the first match occurrence.
- The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

PYTHON

RegEx

Example:

- Print the string passed into the function:

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

RegEx

Example:

- Print the part of the string where there was a match.
- The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

Note: If there is no match, the value **None** will be returned, instead of the Match Object.

PYTHON

PIP

What is PIP?

- PIP is a package manager for Python packages, or modules if you like.

What is a Package?

- A package contains all the files you need for a module.
- Modules are Python code libraries you can include in your project.

PYTHON

PIP

Check if PIP is Installed:

- Navigate your command line to the location of Python's script directory, and type the following:

Check PIP version:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip --version
```

PYTHON

PIP

Install PIP:

- If you do not have PIP installed, you can download and install it from this page: <https://pypi.org/project/pip/>

Download a Package:

- Downloading a package is very easy.
- Open the command line interface and tell PIP to download the package you want.

PYTHON

PIP

- Navigate your command line to the location of Python's script directory, and type the following:

Example:

- Download a package named "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip install camelcase
```

- Now you have downloaded and installed your first package!

PYTHON

PIP

Using a Package:

- Once the package is installed, it is ready to use.
- Import the "camelcase" package into your project.

Example:

- Import and use "camelcase":

```
import camelcase

c = camelcase.CamelCase()

txt = "hello world"

print(c.hump(txt))
```

PYTHON

PIP

Find Packages:

- Find more packages at <https://pypi.org/>.

Remove a Package:

- Use the **uninstall** command to remove a package:

Example:

- Uninstall the package named "camelcase":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip uninstall camelcase
```

PYTHON

PIP

- The PIP Package Manager will ask you to confirm that you want to remove the camelcase package:

```
Uninstalling camelcase-0.2.1:  
Would remove:  
c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-packages\camelcase-0.2-py3.6.egg-info  
c:\users\Your Name\appdata\local\programs\python\python36-32\lib\site-packages\camelcase\*  
Proceed (y/n)?
```

- Press **y** and the package will be removed.

PYTHON

PIP

List Packages:

- Use the **list** command to list all the packages installed on your system:

Example:

- List installed packages:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip list
```

PYTHON

PIP

Result:

Package	Version
<hr/>	
camelcase	0.2
mysql-connector	2.1.6
pip	18.1
pymongo	3.6.1
setuptools	39.0.1

PYTHON

Try....Except

- Error in Python can be of two types i.e. Syntax errors and Exceptions.

Some of the common Exception Errors are :

- **IOError:** if the file can't be opened.
- **KeyboardInterrupt:** when an unrequired key is pressed by the user.
- **ValueError:** when the built-in function receives a wrong argument.
- **EOFError:** if End-Of-File is hit without reading any data.
- **ImportError:** if it is unable to find the module.

PYTHON

Try....Except

- Try and Except statement is used to handle these errors within our code in Python.
 - The **try** block lets you test a block of code for errors.
 - The **except** block lets you handle the error.
 - The **else** block lets you execute code when there is no error.
 - The **finally** block lets you execute code, regardless of the result of the try- and except blocks.

PYTHON

Try....Except

Exception Handling:

- When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
- These exceptions can be handled using the **try** statement:

PYTHON

Try....Except

Example:

- The **try** block will generate an exception, because x is not defined:



A screenshot of a dark-themed code editor window. At the top left, there are three small circular icons. Below them, the code is displayed in green and yellow text. The code consists of a **try** block followed by a **print(x)** statement, and an **except** block with a **print("An exception occurred")** statement.

```
...  
try:  
    print(x)  
except:  
    print("An exception occurred")
```

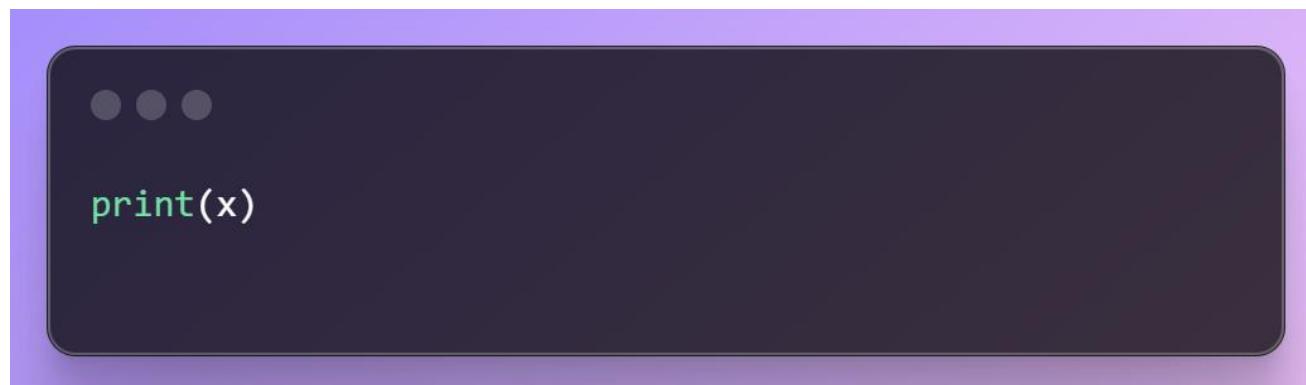
PYTHON

Try....Except

- Since the try block raises an error, the except block will be executed.
- Without the try block, the program will crash and raise an error:

Example:

- This statement will raise an error, because `x` is not defined:

A screenshot of a Python code editor. The code window has a dark background with a light purple header bar. In the code area, there is a dark grey rounded rectangle containing three grey dots at the top left. Below the dots, the word "print" is written in green, followed by "(" and "x" in red. The rest of the code window is empty and dark.

Try....Except

Many Exceptions:

- You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example:

- Print one message if the try block raises a **NameError** and another for other errors:

```
try:  
    print(x)  
except NameError:  
    print("Variable x is not defined")  
except:  
    print("Something else went wrong")
```

Try....Except

Else:

- You can use the **else** keyword to define a block of code to be executed if no errors were raised:

Example:

- In this example,
the **try** block does not
generate any error:

```
...  
try:  
    print("Hello")  
except:  
    print("Something went wrong")  
else:  
    print("Nothing went wrong")
```

Try....Except

Finally:

- The **finally** block, if specified, will be executed regardless if the try block raises an error or not.

```
...  
  
try:  
    print(x)  
except:  
    print("Something went wrong")  
finally:  
    print("The 'try except' is finished")
```

- This can be useful to close objects and clean up resources:

PYTHON

Try....Except

Example:

Try to open and write to a file that is not writable:

- The program can continue, without leaving the file object open.

```
...  
try:  
    f = open("demofile.txt")  
    try:  
        f.write("Lorum Ipsum")  
    except:  
        print("Something went wrong when writing to the file")  
    finally:  
        f.close()  
except:  
    print("Something went wrong when opening the file")
```

Try....Except

Raise an exception:

- As a Python developer you can choose to throw an exception if a condition occurs.
- To throw (or raise) an exception, use the **raise** keyword.

Example:

Raise an error and stop the program if x is lower than 0:

```
...  
x = -1  
  
if x < 0:  
    raise Exception("Sorry, no numbers below zero")
```

PYTHON

Try....Except

- The `raise` keyword is used to raise an exception.
- You can define what kind of error to raise, and the text to print to the user.

Example:

Raise a `TypeError` if `x` is not an integer:

```
...  
x = "hello"  
  
if not type(x) is int:  
    raise TypeError("Only integers are allowed")
```

PYTHON

User Input

- Python allows for user input.
- That means we are able to ask the user for input.
- The method is a bit different in Python 3.6 than Python 2.7.
- Python 3.6 uses the `input()` method.
- Python 2.7 uses the `raw_input()` method.

PYTHON

User Input

- The following example asks for the username, and when you entered the username, it gets printed on the screen:

Python 3.6



A screenshot of a Python code editor window. The code is displayed in a dark-themed interface with a light purple header bar. The code itself is white on a dark background. It contains three gray circular progress indicators at the top left. The code is as follows:

```
username = input("Enter username:")
print("Username is: " + username)
```

PYTHON

User Input

Python 2.7

A dark-themed code editor window with a light purple header bar. In the top-left corner of the header bar are three small gray dots. The main area of the window is dark gray and contains the following Python code:

```
username = raw_input("Enter username:")
print("Username is: " + username)
```

- Python stops executing when it comes to the `input()` function, and continues when the user has given some input.

String Formatting

- To make sure a string will display as expected, we can format the result with the `format()` method.

String `format()`:

- The `format()` method allows you to format selected parts of a string.
- Sometimes there are parts of a text that you do not control, maybe they come from a database, or user input?
- To control such values, add placeholders (curly brackets `{}`) in the text, and run the values through the `format()` method:

String Formatting

Example:

- Add a placeholder where you want to display the price:

```
...
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

String Formatting

- You can add parameters inside the curly brackets to specify how to convert the value:

Example:

- Format the price to be displayed as a number with two decimals:

```
txt = "The price is {:.2f} dollars"
```

- Check out all formatting types in our [String format\(\) Reference](#).

String Formatting

Multiple Values:

- If you want to use more values, just add more values to the `format()` method:

```
...  
quantity = 3  
itemno = 567  
price = 49  
myorder = "I want {} pieces of item number {} for {:.2f} dollars."  
print(myorder.format(quantity, itemno, price))
```

String Formatting

Index Numbers:

- You can use index numbers (a number inside the curly brackets `{0}`) to be sure the values are placed in the correct placeholders:

```
...
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

String Formatting

- Also, if you want to refer to the same value more than once, use the index number:

```
...  
age = 36  
name = "John"  
txt = "His name is {1}. {1} is {0} years old."  
print(txt.format(age, name))
```

String Formatting

Named Indexes:

- You can also use named indexes by entering a name inside the curly brackets `{carname}`, but then you must use names when you pass the parameter values `txt.format(carname = "Ford")`:

```
myorder = "I have a {carname}, it is a {model}."  
print(myorder.format(carname = "Ford", model = "Mustang"))
```

File Handling

- File handling is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.

File Handling

File Handling:

- The key function for working with files in Python is the `open()` function.
- The `open()` function takes two parameters; *filename*, and *mode*.
- There are four different methods (modes) for opening a file:

File Handling

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist.
- "a" - Append - Opens a file for appending, creates the file if it does not exist.
- "w" - Write - Opens a file for writing, creates the file if it does not exist.
- "x" - Create - Creates the specified file, returns an error if the file exists.
- In addition you can specify if the file should be handled as binary or text mode.
- "t" - Text - Default value. Text mode.
- "b" - Binary - Binary mode (e.g. images)

PYTHON

File Handling

Syntax:

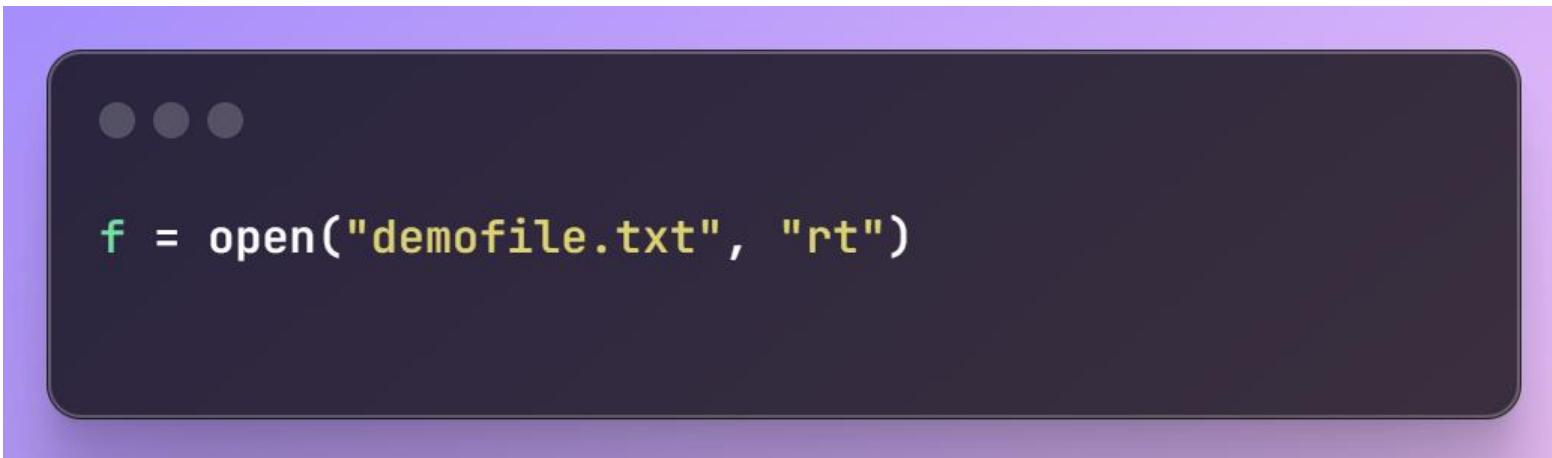
- To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

PYTHON

File Handling

- The code above is the same as:

A screenshot of a terminal window with a dark background and light-colored text. At the top left, there are three small gray dots. Below them, the code `f = open("demofile.txt", "rt")` is displayed in white. The entire terminal window is set against a purple rectangular background.

- Because "r" for read, and "t" for text are the default values, you do not need to specify them.
- **Note:** Make sure the file exists, or else you will get an error.

PYTHON

Read Files

Open a File on the Server:

- Assume we have the following file, located in the same folder as Python:

demofile.txt

Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!

Read Files

- To open the file, use the built-in `open()` function.
- The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

PYTHON

Read Files

- If the file is located in a different location, you will have to specify the file path, like this:

Example:

- Open a file on a different location:

```
f = open("D:\\myfiles\\welcome.txt", "r")
print(f.read())
```

Read Files

Read Only Parts of the File:

- By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example:

- Return the 5 first characters of the file:

```
...  
f = open("demofile.txt", "r")  
print(f.read(5))
```

PYTHON

Read Files

Read Lines:

- You can return one line by using the **readline()** method:

Example:

- Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

PYTHON

Read Files

- By calling `readline()` two times, you can read the two first lines:

Example:

- Read two lines of the file:

```
...
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

PYTHON

Read Files

- By looping through the lines of the file, you can read the whole file, line by line:

Example:

- Loop through the file line by line:

```
...
f = open("demofile.txt", "r")
for x in f:
    print(x)
```

PYTHON

Read Files

Close Files:

- It is a good practice to always close the file when you are done with it.

Example:

- Close the file when you are finish with it:

```
...  
f = open("demofile.txt", "r")  
print(f.readline())  
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write/Create Files

Write to an Existing File:

- To write to an existing file, you must add a parameter to the open() function:
 - "**a**" - Append - will append to the end of the file
 - "**w**" - Write - will overwrite any existing content

Write/Create Files

Example:

- Open the file "demofile2.txt" and append content to the file:

```
• • •  
f = open("demofile2.txt", "a")  
f.write("Now the file has more content!")  
f.close()  
  
#open and read the file after the appending:  
f = open("demofile2.txt", "r")  
print(f.read())
```

PYTHON

Write/Create Files

Example:

- Open the file "demofile3.txt" and overwrite the content :

```
• • •  
f = open("demofile3.txt", "w")  
f.write("Woops! I have deleted the content!")  
f.close()
```

```
#open and read the file after the overwriting:  
f = open("demofile3.txt", "r")  
print(f.read())
```

Note: the "**w**" method will overwrite the entire file.

PYTHON

Write/Create Files

Create a New File:

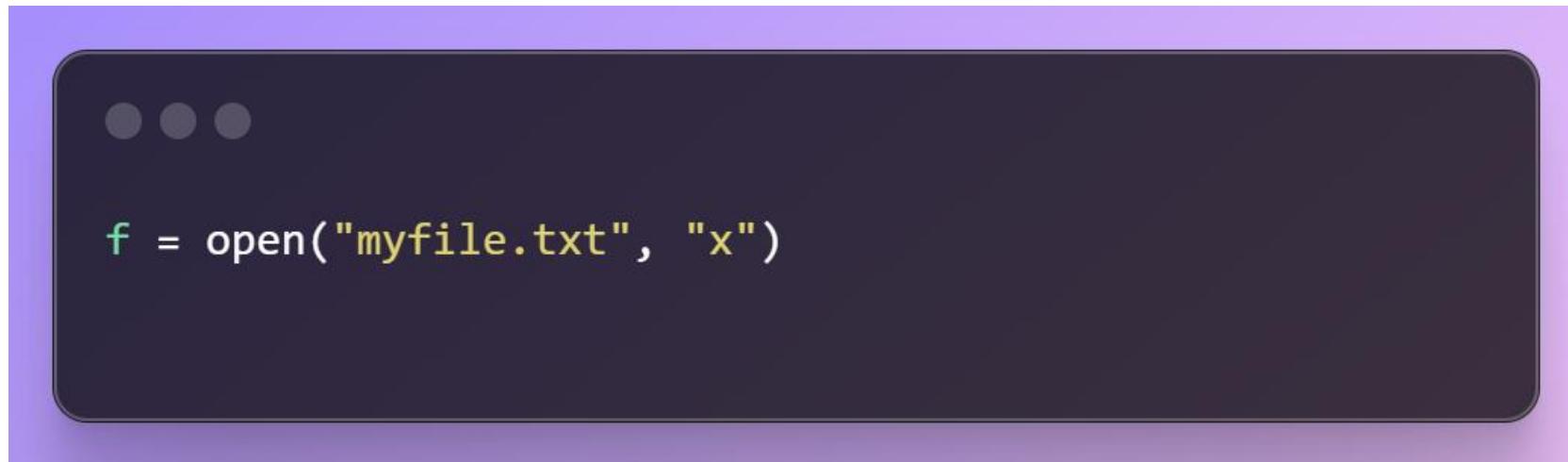
- To create a new file in Python, use the `open()` method, with one of the following parameters:
 - `"x"` - Create - will create a file, returns an error if the file exist
 - `"a"` - Append - will create a file if the specified file does not exist
 - `"w"` - Write - will create a file if the specified file does not exist

PYTHON

Write/Create Files

Example:

- Create a file called "myfile.txt":



A screenshot of a terminal window with a dark background and light-colored text. At the top left, there are three gray dots. Below them, the code `f = open("myfile.txt", "x")` is displayed in white. The terminal window has a black border and is set against a purple gradient background.

Result: a new empty file is created!

PYTHON

Write/Create Files

Example:

- Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

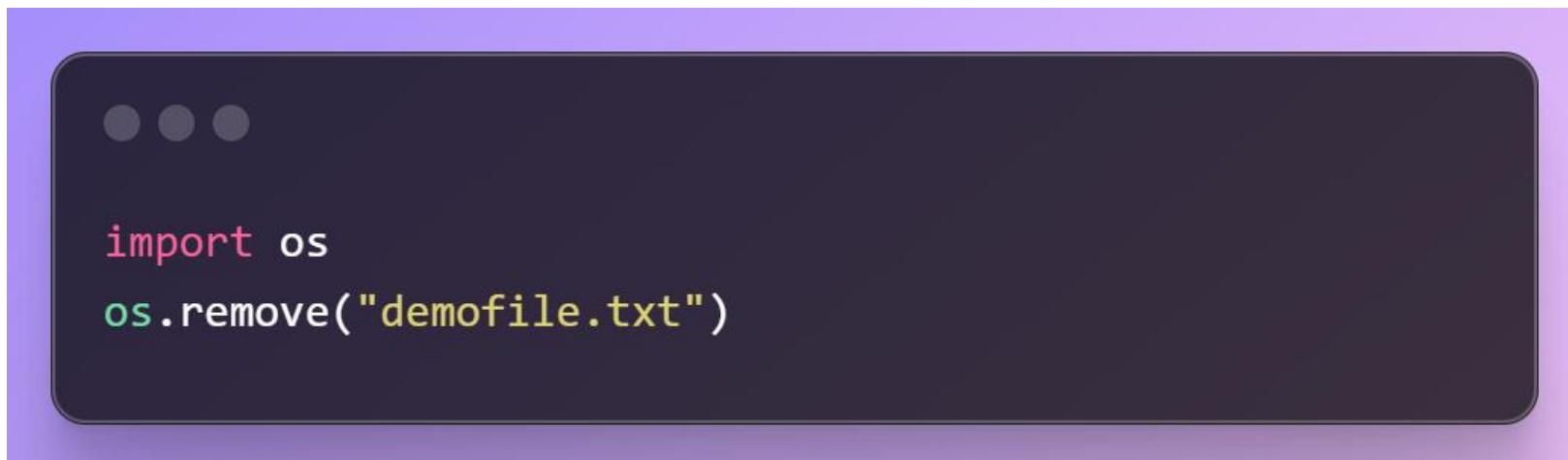
Delete Files

Delete a File:

- To delete a file, you must import the OS module, and run its `os.remove()` function:

Example:

- Remove the file "demofile.txt":



A screenshot of a terminal window with a dark background and light-colored text. At the top left, there are three small circular icons. Below them, the text "import os" is written in pink, followed by "os.remove("demofile.txt")" in green. The entire terminal window is set against a purple background.

```
import os
os.remove("demofile.txt")
```

Delete Files

Check if File exist:

- To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example : Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exist")
```

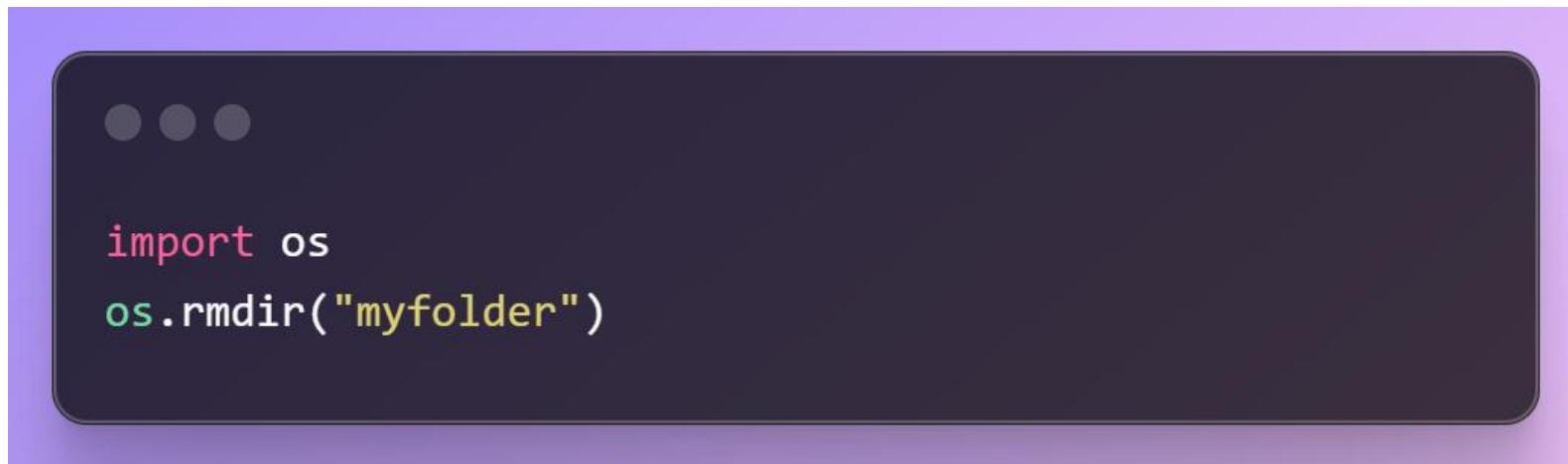
Delete Files

Delete Folder:

- To delete an entire folder, use the `os.rmdir()` method:

Example:

- Remove the folder "myfolder":



```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.