Basics

Syntax for	How to use	Explained

Comments	`# print(1 + 2)`
	`print(5 * 10)`
	`# This program will only print 50`
		We call the sequence of characters that follows the `#` a code comment; any code that follows `#` will not be executed

Arithmetical Operations	`1 + 2  # output: 3`	Addition
	`4 - 5  # output: -1`	Substraction
	`30 * 2  # output: 60`	Multiplication
	`20 / 3  # output: 6.666666666666667`	Division
	`4 ** 3  # output: 64`	Exponentiation
	`(4 * 18) ** 2 / 10  # output: 518.4`	Use parentheses to control the order of operations

Initializing Variables	`cost = 20`
	`total_cost = 20 + 2**5`
	`currency = 'USD'`
	`1_app = 'Facebook' # this will cause an error`
		Variable names can only contain letters, numbers, and underscores—they cannot begin with a number

Updating Variables	`x = 30`
	`print(x)  # 30 is printed`
	`x = 50`
	`print(x)  # 50 is printed`
		To update a variable, use the `=` assignment operator to set a new value

Syntax for	How to use	Explained

Operation Shortcuts	`x += 2  # Addition`
	`x -= 2  # Subtraction`
	`x *= 2  # Multiplication`
	`x /= 2  # Division`
	`x **= 2  # Exponentiation`
		Augmented assignment operators: used to update a variable in place without repeating the variable name; for example, instead of writing: `x = x + 2`, you can use: `x += 2`

Data Types	`x = [1, 2, 3]`
	`y = 4`
	`print(type(x))  # list`
	`print(type(y))  # integer`
	`print(type('4'))  # string`
		Use the `type()` command to determine the data type of a value or variable.

Converting Data Types	`int('4')  # casting a string to an integer`
	`str(4)  # casting an integer as a string`
	`float('4.3')  # casting a string as a float`
	`str(4.3)  # casting a float as a string`
		Converting between data types is also referred to as **casting**

Data Structures

Syntax for	How to use	Explained

Lists	`a_list = [1, 2]`
	`a_list.append(3)`
	`print(a_list)  # output: [1, 2, 3]`
		Creating a list and appending a value to it

	`row_1 = ['Facebook', 0.0, 'USD', 2974676]` `row_2 = ['Instagram', 4.5, 'USD', 2161558]`	Creating a list of data points; lists can store multiple data types at the same time

Indexing	`print(row_1[0])  # output: 'Facebook'`
	`print(row_2[1])  # output: 4.5`
	`print(row_1[3])  # output: 2974676`
	`print(row_2[3])  # output: 2161558`
		Retrieving an element from a list using each item's index number; note that list indexing begins at `0`

Negative Indexing	`print(row_1[-1])  # output: 2974676`
	`print(row_2[-1])  # output: 2161558`
	`print(row_1[-3])  # output: 0.0`
	`print(row_2[-4])  # output: 'Instagram'`
		Negative list indexing works by counting backwards from the last element, beginning with `-1`

	`num_ratings = [row_1[-1], row_2[-1]]`
	`print(name_and_ratings)`
	`# output: [2974676, 2161558]`
		Retrieving multiple list elements to create a new list

List Slicing	`row_3 = ['Clash of Clans', 0.0,`
	`  'USD', 2130805, 4.5]`
	`print(row_3[:2])`
	`# output: ['Clash of Clans', 0.0]`
	`print(row_3[1:4])`
	`# output: [0.0, 'USD', 2130805]`
	`print(row_3[3:])`
	`# output: [2130805, 4.5]`
		List slicing includes the start index but **excludes** the end index; when the start is omitted, the slice begins at the start of the list; when the end is omitted, it continues to the end of the list

Syntax for	How to use	Explained

List of Lists	`from csv import reader`
	`opened_file = open('AppleStore.csv')`
	`read_file = reader(opened_file)`
	`apps_data = list(read_file)`
		Opening a dataset file and using it to create a list of lists

	`row_1 = ['Facebook', 'USD', 2974676, 3.5]`
	`row_2 = ['Instagram', 'USD', 2161558, 8.2]`
	`row_3 = ['Clash', 0.0, 'USD', 2130805, 4.5]`
	`row_4 = ['Fruit', 1.99, 'USD', 698516, 9.1]`
	
	`lists = [row_1, row_2, row_3, row_4]`
		Creating a list of lists by initializing a new list whose elements are themselves lists

Indexing	`first_row_first_element = lists[0][0]`
	`# output: 'Facebook'`
	`second_row_third_element = lists[1][2]`
	`# output: 2161558`
	`third_row_last_element = lists[-2][4]`
	`# output: 4.5`
	`last_row_last_element = lists[-1][-1]`
	`# output: 9.1`
		Retrieving an element from a list of lists by first selecting the row, then the element within that row

Slicing List of Lists	`first_two_rows = lists[:2]`
	`last_two_rows = lists[-2:]`
	`all_but_first_row = lists[1:]`
	`second_row_partial = lists[1][:3]`
	`# output: ['Instagram', 'USD', 2161558]`
	`last_row_partial = lists[-1][1:-2]`
	`# output: [1.99, 'USD']`
		Slicing lists of lists allows extracting full rows or specific elements from a single row; positive indices select from the start, and negative indices select from the end

Data Structures

Syntax for	How to use	Explained

Dictionaries	`# First way`
	`dictionary = {'key_1': 1, 'key_2': 2}`
	`# Second way`
	`dictionary = {}`
	`dictionary['key_1'] = 1`
	`dictionary['key_2'] = 2`
		Creating a dictionary by defining **key:value** pairs at time of initialization (first way) or by creating an empty dictionary and setting the value for each key (second way)

	`dictionary = {'key_1': 100, 'key_2': 200}`
	`dictionary['key_1']  # outputs 100`
	`dictionary['key_2']  # outputs 200`
		Retrieve individual dictionary values by specifying the key; keys can be strings, numbers, or tuples, but not lists or sets

	`dictionary = {'key_1': 100, 'key_2': 200}`
	`'key_1' in dictionary  # outputs True`
	`'key_5' in dictionary  # outputs False`
	`100 in dictionary  # outputs False`
		Use the `in` operator to check for dictionary key membership

	`dictionary = {'key_1': 100, 'key_2': 200}`
	`dictionary['key_1'] += 600`
	`dictionary['key_2'] = 400`
	`print(dictionary)`
	`# output: {'key_1': 700, 'key_2': 400}`
		Update dictionary values by specifying the key and assigning a new value

Syntax for	How to use	Explained

Frequency Tables	`frequency_table = {}`
	`for row in a_data_set:`
	`  a_data_point = row[5]`
	`  if a_data_point in frequency_table:`
	`  frequency_table[a_data_point] += 1`
	`  else:`
	`  frequency_table[a_data_point] = 1`
		Builds a frequency table by counting occurrences of values in the 6th column (`row[5]`) of `a_data_set`, incrementing the count if the value exists, or adding it if not

Defined Intervals	`data_sizes = {'0 - 10 MB': 0,`
	`  '10 - 50 MB': 0,`
	`  '50 - 100 MB': 0,`
	`  '100 - 500 MB': 0,`
	`  '500 MB +': 0}`
	`for row in apps_data[1:]:`
	`  data_size = float(row[2])`
	`  if data_size <= 10000000:`
	`  data_sizes['0 - 10 MB'] += 1`
	`  elif 10000000 < data_size <= 50000000:`
	`  data_sizes['10 - 50 MB'] += 1`
	`  elif 50000000 < data_size <= 100000000:`
	`  data_sizes['50 - 100 MB'] += 1`
	`  elif 100000000 < data_size <= 500000000:`
	`  data_sizes['100 - 500 MB'] += 1`
	`  elif data_size > 500000000:`
	`  data_sizes['500 MB +'] += 1`
		Categorizes app sizes from `apps_data` into predefined ranges (e.g., `'0 - 10 MB'`) and increments the corresponding count based on each app's size inside the `data_sizes` dictionary

Functions

Syntax for	How to use	Explained

Basic Functions	`def square(number):`
	`  return number**2`
	
	`print(square(5))  # output: 25`
		Create a function with a single parameter: `number`

	`def add(x, y):`
	`  return x + y`
	
	`print(add(3, 14))  # output: 17`
		Create a function with more than one parameter `x` and `y`

	`def freq_table(list_of_lists, index):`
	`  frequency_table = {}`
	`  for row in list_of_lists:`
	`  value = row[index]`
	`  if value in frequency_table:`
	`  frequency_table[value] += 1`
	`  else:`
	`  frequency_table[value] = 1`
	`  return frequency_table`
		This function creates a frequency table for any given column `index` of the provided `list_of_lists`

Syntax for	How to use	Explained

Arguments	`def subtract(a, b):`
	`  return a - b`
	
	`print(subtract(a=10, b=7))  # output: 3`
	`print(subtract(b=7, a=10))  # output: 3`
	`print(subtract(10, 7))  # output: 3`
		Use named arguments and positional arguments

Helper Functions	`def find_sum(lst):`
	`  a_sum = 0`
	`  for element in lst:`
	`  a_sum += float(element)`
	`  return a_sum`
	
	`def find_length(lst):`
	`  length = 0`
	`  for element in lst:`
	`  length += 1`
	`  return length`
	
	`def mean(lst):`
	`  return find_sum(lst) / find_length(lst)`
	
	`print(mean([1, 2, 4, 6, 2]))  # output: 3`
		Define **helper functions** to find the sum and length of a list; the `mean` function reuses these to calculate the average by dividing the sum by the length

Functions

Syntax for	How to use	Explained

Multiple Arguments	`def price(item, cost):`
	`  return "The " + item + " costs $" +`
	`  str(cost) + "."`
	
	`print(price("chair", 40.99))`
	`# output: 'The chair costs $40.99.'`
		Define a function that accepts multiple arguments and returns a formatted string combining both inputs

	`def price(item, cost):`
	`  print("The " + item + " costs $" +`
	`  str(cost) + ".")`
	
	`price("chair", 40.99)`
	`# output: 'The chair costs $40.99.'`
		Similar to the previous function, but uses `print()` to display the string immediately rather than returning it for further use

Default Arguments	`def add_value(x, constant=3.14):`
	`  return x + constant`
	
	`print(add_value(6, 3))  # output: 9`
	`print(add_value(6))  # output: 9.14`
		Define a function with a default argument; if no second argument is provided, the default value is used in the calculation

Syntax for	How to use	Explained

Multiple Return Statements	`def sum_or_difference(a, b, return_sum=True):`
	`  if return_sum:`
	`  return a + b`
	`  else:`
	`  return a - b`
	
	`print(sum_or_difference(10, 7))`
	`# output: 17`
	`print(sum_or_difference(10, 7, False))`
	`# output: 3`
		This function uses multiple return statements to either return the sum or the difference of two values, depending on the `return_sum` argument, which defaults to `True`

	`def sum_or_difference(a, b, return_sum=True):`
	`  if return_sum:`
	`  return a + b`
	`  return a - b`
	
	`print(sum_or_difference(10, 7))`
	`# output: 17`
	`print(sum_or_difference(10, 7, False))`
	`# output: 3`
		This function is similar to the previous one but omits the `else` clause, returning the difference directly when `return_sum` is `False`, simplifying the logic

Returning Multiple Values	`def sum_and_difference(a, b):`
	`  a_sum = a + b`
	`  difference = a - b`
	`  return a_sum, difference`
	
	`sum_1, diff_1 = sum_and_difference(15, 10)`
		This function returns multiple values (sum and difference) at once by separating them with commas, allowing them to be unpacked into separate variables when called

Strings

Syntax for	How to use	Explained
Formatting	`continents = "France is in {} and China is in {}".format("Europe", "Asia")` `# France is in Europe and China is in Asia`	Insert values by order into placeholders for simple string formatting
	`squares = "{0} times {0} equals {1}".format(3,9)` `# 3 times 3 equals 9`	Use indexed placeholders to repeat or position values
	`population = "{name}'s population is {pop} million".format(name="Brazil", pop=209)` `# Brazil's population is 209 million`	Assign values to named placeholders using variable names
	`two_decimal_places = "I own {:.2f}% of the company".format(32.5548651132)` `# I own 32.55% of the company`	Format a float to two decimal places for precise output
	`india_pop = "The approx pop of {} is {:,}".format("India", 1324000000)` `# The approx pop of India is 1,324,000,000`	Insert a number with commas as a thousand separator by position
	`balance_string = "Your bank balance is ${:,.2f}".format(12345.678)` `# Your bank balance is $12,345.68`	Format a number with commas and two decimal places for currency formatting

Syntax for	How to use	Explained
String Cleaning	`green_ball = "red ball".replace("red", "green")` `# green ball`	Replace parts of a string by specifying the old and new values
	`friend_removed = "hello there friend!".replace(" friend", "")` `# hello there!`	Remove a specified substring from a string by replacing it with an empty string

	`bad_chars = ["'", ",", ".", "!"]`
	`string = "We'll remove apostrophes, commas, periods, and exclamation marks!"`
	`for char in bad_chars:`
	`  string = string.replace(char, "")`
	`# Well remove apostrophes commas periods and`
	`# exclamation marks`
		Use a loop to remove multiple specified characters from a string by replacing them with an empty string

	`print("hello, my friend".title())` `# Hello, My Friend`	Capitalize the first letter of each word in the string
	`split_on_dash = "1980-12-08".split("-")` `# ['1980', '12', '08']`	Split a string into a list of substrings based on the specified delimiter
	`first_four_chars = "This is a long string."[:4]` `# This`	Slice the string to return the first four characters; missing indices default to the start or end of the string
	`superman = "Clark" + " " + "Kent"` `# Clark Kent`	Concatenate strings using the `+` operator to join them with a space

Control Flow

Syntax for	How to use	Explained

For Loops	`row_1 = ['Facebook', 0.0, 'USD', 2974676]`
	`for element in row_1:`
	`  print(element)`
		With each iteration, this loop will print an element from `row_1`, in order

	`rating_sum = 0`
	`for row in apps_data[1:]:`
	`  rating = float(row[7])`
	`  rating_sum = rating_sum + rating`
		Convert a column of strings (`row[7]`) in a list of lists (`apps_data`) to a float and keep a running sum of ratings

	`apps_names = []`
	`for row in apps_data[1:]:`
	`  name = row[1]`
	`  apps_names.append(name)`
		Append values with each iteration of a `for` loop

Conditional Statements	`price = 0`
	`print(price == 0)  # Outputs True`
	`print(price == 2)  # Outputs False`
		Use comparison operators to check if a value equals another, returning `True` or `False`

	`print('Games' == 'Music')  # Outputs False`
	`print('Games' != 'Music')  # Outputs True`
	`print([1,2,3] == [1,2,3])  # Outputs True`
	`print([1,2,3] == [1,2,3,4])  # Outputs False`
		Compare strings and lists using `==` for equality and `!=` for inequality, returning `True` or `False`

Syntax for	How to use	Explained
If Statements	`if True:` `  print('This will always be printed.')`	The condition `True` always executes the code inside the `if` block

	`if True:`
	`  print(1)`
	`if 1 == 1:`
	`  print(2)`
	`  print(3)`
		Both conditions evaluate to `True`, so all print statements are executed

	`if True:`
	`  print('First Output')`
	`if False:`
	`  print('Second Output')`
	`if True:`
	`  print('Third Output')`
		Only the blocks with `True` conditions are executed, so the second print statement is skipped

Else Statements	`if False:`
	`  print(1)`
	`else:`
	`  print('The condition above was false.')`
		The code in the `else` clause is always executed when the if statement is `False`

Control Flow

Syntax for	How to use	Explained

Else Statements	`if "car" in "carpet":`
	`  print("The substring was found.")`
	`else:`
	`  print("The substring was not found.")`
		The `in` operator checks if a substring exists in a string, executing the corresponding `if` or else block

Elif Statements	`if 3 == 1:`
	`  print('3 does not equal 1.')`
	`elif 3 < 1:`
	`  print('3 is not less than 1.')`
	`else:`
	`  print('Both conditions above are false.')`
		The `elif` statement allows for multiple conditions to be tested; if the `if` condition is `False`, the `elif` condition is checked, and if both are `False`, the `else` block is executed

Multiple Conditions	`if 3 > 1 and 'data' == 'data':`
	`  print('Both conditions are true!')`
	`if 10 < 20 or 4 >= 5:`
	`  print('At least one condition is true.')`
		Use `and` to require both conditions to be `True` and `or` to require at least one condition to be `True`

	`if (20 > 3 and 2 != 1) or 'Games' == 'Game':` `  print('At least one condition is true.')`	Use parentheses to group conditions and control the order of evaluation in complex logical expressions

Object-Oriented Programming Basics

Syntax for	How to use	Explained
Defining Classes	`class MyClass:` `  pass`	Define an empty class

Instantiating Class Objects	`class MyClass:`
	`  pass`
	`mc_1 = MyClass()`
		Instantiate an object from the class by calling the class name followed by parentheses

Setting Class Attributes	`class MyClass:`
	`  def __init__(self, param_1):`
	`  self.attribute = param_1`
	`mc_2 = MyClass("arg_1")`
	`# mc_2.attribute is set to "arg_1"`
		Use the `__init__` method to initialize an object's attributes during instantiation by passing arguments

Defining Class Methods	`class MyClass:`
	`  def __init__(self, param_1):`
	`  self.attribute = param_1`
	`  def add_20(self):`
	`  self.attribute += 20`
	`mc_3 = MyClass(10)  # mc_3.attribute is 10`
	`mc_3.add_20()  # mc_3.attribute is now 30`
		Define a method within the class to modify an attribute; `add_20` increases the value of `attribute` by 20 when called

Dates and Time

Syntax for	How to use	Explained
Importing Datetime Examples	`import datetime` `current_time = datetime.datetime.now()`	Import the module, requiring the full path to access functions or classes
	`import datetime as dt` `current_time = dt.datetime.now()`	Import the module with alias `dt` for shorter references, a common practice
	`from datetime import datetime` `current_time = datetime.now()`	Import only the `datetime` class, enabling direct access without the module name prefix

	`from datetime import datetime, date`
	`current_time = datetime.now()`
	`current_date = date.today()`
		Import multiple classes from the module, allowing direct use of their respective methods

	`from datetime import *`
	`current_time = datetime.now()`
	`current_date = date.today()`
	`min_year = MINYEAR`
	`max_year = MAXYEAR`
		Import all classes and functions, making every definition and all constants accessible without using a prefix; this is not advised for this module

Creating Datetime Objects	`import datetime as dt` `eg_1 = dt.datetime(1985, 3, 13, 14, 30, 45)`	Create a `datetime` object with both date (March 13, 1985) and time (14:30:45) components

	`from datetime import datetime as dt`
	`eg_2 = dt.strptime("15/08/1990 08:45:30",`
	`"%d/%m/%Y %H:%M:%S")`
		Convert a formatted string into a `datetime` object; the "p" stands for parsing

Creating Datetime Objects	`eg_2_str = eg_2.strftime(`
	`"%B %d, %Y at %I:%M %p")`
	`# "August 15, 1990 at 08:45 AM"`
		Convert a `datetime` object into a formatted string; the "f" stands for formatting

	`eg_3 = dt.time(hour=5, minute=23,`
	`second=45, microsecond=123456)`
	`# 05:23:45.123456`
		Create a `time` object that includes microseconds

	`eg_4 = dt.timedelta(weeks=3)`
	`future_date = eg_1 + eg_4`
	`# 1985-04-03 14:30:45`
		Add a `timedelta` object representing 3 weeks to a `datetime` object to calculate a future date

Accessing Datetime Attributes	`eg_1.year  # returns 1985`
	`eg_1.month  # returns 3`
	`eg_2.day  # returns 15`
	`eg_2.hour  # returns 8`
	`eg_3.minute  # returns 23`
	`eg_3.microsecond  # returns 123456`
		Access specific components directly from `datetime` and `time` objects using their built-in attributes

	`eg_2_time = eg_2.time()` `# 08:45:30`	Extract the time component from a `datetime` object that contains both date and time using the `.time()` method