```python
# From Python Full Course for Beginners
# Programming with Mosh
# https://www.youtube.com/watch?v=_uQrJ0TkZlc

# Math operators: This is an example of using math operators
# to calculate a buyer's interest rate on a house if they have
# good credit vs bad credit.

house_price = 1000000
has_good_credit = True
good_rate = .10
lesser_rate = .20

if has_good_credit:
    down_payment = (house_price * good_rate)
else:
    down_payment = (house_price * lesser_rate)

print(f"Down payment = ${down_payment}.")

# ........................................................................ #
# 08-19-22
# CONDITIONS in PYTHON (1:06:41)

# If an applicant has high income and good credit, they are eligible
# for a loan: LOGICAL AND OPERATOR

has_high_income = True
has_good_credit = True

if has_high_income and has_good_credit:  # If both these are True
    print("first AND: Eligible for loan.")

has_high_income = False
has_good_credit = True

if has_high_income and has_good_credit:  # Since BOTH are not true,
    print("second AND: Eligible for loan.")  # will not print eligible.

# Use OR operator to have a conditional based on ONE of the conditions
# being true.

has_high_income = True
has_good_credit = False
# Will only not print if BOTH conditions are false
if has_high_income or has_good_credit:  # If one of these is True
    print("OR conditional: Eligible for loan.")

# AND: Both conditions must be true
# OR: At lease one of the conditions must be true
# NOT: converts boolean value to false
# If applicant has good credit and DOES NOT have a criminal record,
# they are eligible for a loan.

has_good_credit = True
has_criminal_record = False
# The criminal record part of the conditional will end up returning
# a True since it is True that they do NOT have a criminal record.
if has_good_credit and not has_criminal_record:
    print("NOT operator: Eligible for loan.")

# ......................................................................... #
# COMPARISON OPERATORS: comparing a variable with a value
# Expression = piece of code that produces a value
# The one below is a boolean expression, because it is based on whether
# or not the expression returns True

temperature = 30
if temperature > 30:
    print("It is a hot day.")
else:
    print("It is not a hot day.")

# ......................................................................... #
# COMPARISON OPERATORS = <, >, <=, >=, ==, !=

# Practice: create a name input field with the requirements that the name
# must be at least 3 characters long, or user gets error message as such,
# name cannot be more than 50 characters long, or error message as such,
# else name looks good.

name = input("Please tell me your name: ")

if len(name) > 2 and len(name) < 51:
    print(f"Nice name, {name}!")
elif len(name) < 3:
    print("Name field must contain at least 3 characters.")
elif len(name) > 50:
    print("No name should be that long.")

# ......................................................................... #
# PROJECT: Weight Conversion

# Ask user for weight
# Ask user if that weight is in lbs or kgs: L for lbs, K for kgs
# Convert the weight to the opposite units and return in print statement

# Float rather than int due to the operations we need to perform with it.
# Could also have done this at the time of input:
# float(input("What is your weight? ")
weight = float(input("What is your weight? "))
units = input("Is that in (type l for) pounds or (type k for) kilos? ").lower()
```

```python

# convert the input to lower so that no matter which they enter, the program
# can figure out what to do.
while units not in ('k', 'l'):
    print("Invalid unit of measurement. Please try again.")
    units = input("Is that in (type l for) pounds or (type k for) kilos? ").lower()
if units == "k":
    # Inputs are always strings, so we need to convert to float,
    conversion = (weight) * 2.2
    print(f"Your converted weight is {conversion:.2f} pounds.")
elif units == 'l':
    conversion = (weight) * 0.45
    print(f"Your converted weight is {conversion:.2f} kilos.")

# ........................................................................ #
# WHILE LOOP: consist of -- while condition: --
# As long as the condition is true, the code inside the while loop will be
# executed.
i = 1
while i < 5:
    print('*' * i)  # Prints a triangle
    i += 1
print("done!")
# ------------------------------------------------------#
# While loop guessing game (simple version)
secret_number = 9
guess_count = 0
guess_limit = 3
while guess_count < guess_limit:
    # We need their guess to be stored as an int
    user_guess = int(input("Guess a number: "))
    guess_count += 1
    if user_guess == secret_number:
        print("You win!")
        # If the user makes the right guess, break out of loop
        break
    else:
        print("Nope! Try again!")
else:
    print("Sorry, no cookie for you!")

# ...................................................................... #
# CAR GAME:

answer = ''
# So that we can catch incidents when the car is already started or stopped
# and the user tries to start or stop again, we need a boolean variable.
car_started = False
# Because we add .lower() after the input in the while loop, it will lower
# case the input every time instead of having to type answer.lower() every
# time we as if answer = something.
while answer != "quit":
    answer = input('> ').lower()
    if answer == "start":
        if car_started:
            print("Car has already been started. What are you thinking?!")
        else:
            # Start car and set the value of car_started to true
            print("You have successfully started the car! VROOM VROOM!!!!")
            car_started = True
    elif answer == "stop":
        if not car_started:
            print("The car is already stopped. There is not much I can do.")
        else:
            # Stop car and set the value of car_started to false
            print("You have successfully stopped the car. Car is waiting patiently.")
            car_started = False

    # We can print a list of commands for if the user asks for help by putting
    # a doc-string, multi-line text into the print function:
    # To avoid to over-indentation you get with a doc-string when printed like
    # this, you can delete the indentation as shown below. Looks weird here,
    # but looks much better in the terminal.

    elif answer == "help":
        print("""
start - to start the car
stop - to stop the car
quit - to quit the game
        """)
    else:
        print("I am sorry, but this car does not understand what you said.")

```

```python
# 08-19-22 - Python Tutorial: Python Full Course for Beginners with Mosh
# (https://youtu.be/_uQrJ0TkZlc) Starting at For Loops (1:41:59)
# ..................................................................................  #
# FOR LOOPS - used to iterate through a collection, like characters in a string,
# items in a list, range of numbers,

for item in 'Python':
    print(item)
#*************************************#
for item in ['John', 'Sarah', 'Jordan', 'Ellen']:
    print(item)
#*************************************#
for number in range(80, 101, 2):
    # args = start, stop + 1, step
    print(number)
#*************************************#
prices = [10, 20, 30, 40]
total = 0
for price in prices:
    total += price
print(f"Purchase total =  ${total:.2f}")
# ..................................................................................  #
# AUGUST 20, 2022
# NESTED LOOPS: creating coordinates that change with a nested loop
for x in range(4):      # Print multiplies the nest repetitions, incrementing x
    for y in range(3):  # Print multiplies the nest repetitions, incrementing y
        print(f"({x},{y})") # So 3 loops for x each time and 2 loops for y

# CHALLENGE: Print an F made of Xs
numbers = [5, 2, 5, 2, 2]
for number in numbers:
    print('X' * number)

# If we could not multiply the printing of X times the number in the list in the loop,
# it will require nested loops:

numbers2 = [5, 2, 5, 2, 2]
for number in numbers2:
    output = ''                 # We have to construct a string with an inner loop
    for count in range(number): # for each number in the list of numbers and then
        output += 'X'           # print line for line after each inner loop.
    print(output)

numbers_l = [2, 2, 2, 2, 6]
for number in numbers_l:
    print('X' * number)

# ..................................................................................  #
# LISTS:

names = ['John', 'Bob', 'Mosh', 'Sarah', 'Mary']
# A list of 5 items that are all strings.
# Each can be accessed by index.
print(names[3])      # Prints Sarah [3]
print(names[2:])     # Prints Mosh [2] to the end
print(names[3][2])  # Prints the r from Sarah [3][2]
print(names[::-1])  # Reverses the entire list
print(names[1][::-1])   # Prints Bob in reverse
print(names[4][::-1])   # Prints Mary in reverse

# Modifying: Remove h from John
names[0] = 'Jon'
print(names)        # Prints with new spelling of Jon

# Write a program to find the largest number in a list

list_numbers = [5, 2, 7, 11, 56, 23, 7, 99, 23, 12]
print(max(list_numbers))

# Or the long way
max = 0
for number in list_numbers:
    if number > max:
        max = number
print(max)

# ........................................................................................ #
# Two Dimensional Lists: A list where each item is another list of a specified number
# In this way, we can create a matrix:

matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix[0][1]) # PRINTS 2, because it is spot [0][1]
# Nested loop to iterate over all items:
for row in matrix:
    for digit in row:
        print(digit)        # PRINTS all numbers, one per line

# ........................................................................................ #
# LIST FUNCTIONS: Functions that can be performed on lists

some_numbers = [5, 2, 1, 7, 4]
some_numbers.append(20)      # <- adds to end of list
print(some_numbers)         # PRINTS: [5, 2, 1, 7, 4, 20]

# HINT: When you type in a method to use and open parentheses, there is a hint or tip
# that pops up above your cursor to show you what arguments that method takes.

some_numbers.insert(3, 23)  # <- insert gets two values, the index and the object to insert
print(some_numbers)         # Now prints: [5, 2, 1, 23, 7, 4, 20]
```

```

some_numbers.remove(5)      # Removes the object you pass to it from the list
print(some_numbers)         # PRINTS: [2, 1, 23, 7, 4, 20]

# some_numbers.clear()        # This does not take any arguments
# print(some_numbers)         # PRINTS: []

some_numbers.pop()          # Removes the last item from the list
print(some_numbers)         # PRINTS: [2, 1, 23, 7, 4]

print(some_numbers.index(23))   # This returns the index of the first occurrence
# ^^ PRINTS: 2                  # of the object passed to it.
# If you check for the an item that is not in the list, you will get a ValueError

# You can also check for the existence of something in a list by using the IN
# operator, and it does not cause an error:

print(50 in some_numbers)       # <- PRINTS: False, because it is not there.

# We can also get the number of occurrences of an object in a list by passing to count.
# First we must add multiple occurrences to our list:
some_numbers.append(23)
some_numbers.append(7)
print(some_numbers.count(23))   # <- PRINTS: 2

# We can sort the list, but it will not return any values. The print statement below
# returns None. It just sorts the list in place. Ascending by default.
print(some_numbers.sort())

# Instead, you can call it to sort the list, and then print the list:
some_numbers.sort()
print(some_numbers)     # <- PRINTS: [1, 2, 4, 7, 7, 23, 23]

some_numbers.reverse()  # To reverse items in a list, descending order.
print(some_numbers)     # <- PRINTS [23, 23, 7, 7, 4, 2, 1]

other_numbers = some_numbers.copy()     # <- This will be a copy of some_numbers
other_numbers.append(22)
print(other_numbers, some_numbers)
# PRINTS: [23, 23, 7, 7, 4, 2, 1, 22] [23, 23, 7, 7, 4, 2, 1]
# Both lists, but only other_numbers, printed first, has had 22 appended to the end

# CHALLENGE: Write a program to remove duplicates in a list:
some_numbers = list(set(some_numbers))
print(some_numbers) # <- Prints: [1, 2, 4, 7, 23] (Duplicates have been removed.)

# LONGER WAY:
uniques = []
for i in other_numbers:
    if i not in uniques:
        uniques.append(i)
print(uniques)              # <- PRINTS: [23, 7, 4, 2, 1, 22] as a new list w/ new name
```

```python
# ...................................................................................... #
# TUPLES: similar to lists and can store items, but cannot be modified
# They are immutable. You can only get information about them.

num_tuple = (1, 2, 3)   # Only has two methods, count to get the count of an item
                  # and index to get the first occurrence of an item.
print(num_tuple[1])     # <- PRINTS: 2
# Tuples are useful when you want a list you can be sure you will not accidentally
# modify.

# UNPACKING
coordinates = (1, 2, 3)
# Variables can be assigned to items from a tuple one at a time:
x = coordinates[0]
y = coordinates[1]
z = coordinates[2]
# But they can also be unpacked:
x, y, z = coordinates   # This is identical to the three lines of code above.
print(x, y, z)  # <- PRINTS: 1 2 3

# UNPACKING also works with LISTS!

# 08-20-22 - Python Tutorial: Python Full Course for Beginners with Mosh
# (https://youtu.be/_uQrJ0TkZlc) Starting at DICTIONARIES (2:18:32)
# ........................................................................................ #
# DICTIONARIES: Used to store information you want to keep in key-value pairs.

# Example: You have a customer named John Smith with many different attributes:
# Name: John Smith                    Key is name, value is John Smith
# Email: JohnSmith@gmail.com        Key is email... and so on
# Phone: 123-456-7890

customer = {
    'name': 'John Smith',
    'age': 30,
    'email': 'JohnSmith@gmail.com',
    'phone': '123-456-7890',
    'verified': 'yes'
}

# Each key in a dictionary must be unique
# Values can be anything: string, number, boolean, list, etc.
# Now, each item in the dictionary can be accessed by key with []

print(customer['name'])  # This is the same as typing "John Smith"
# If you try to pass a key that does not exist, you will get a key error.
# Keys are also case-sensitive.

# You can also use the get method for dictionaries:
# And if you want to use a key/value that is not already in the dictionary, you can do
# so and supply a default value at the same time.
print(customer.get('name'))
print(customer.get('birthday', "Jan 1, 1980"))  # Does not add this to dictionary, however.

# UPDATING:
customer['name'] = "Jack Smith"
customer['birthday'] = "Jan 1, 1980"
print(customer['name'], customer['birthday'])  # <- Now, John is Jack, and the dictionary
# ^^^ PRINTS: Jack Smith Jan 1, 1980                    contains his birthday.

# CHALLENGE: Write a program that takes a phone number and prints it out in words

phone = input("Phone: ")

for index, number in enumerate(phone):
    p_num = {
        "0": 'zero ',
        "1": 'one ',
        "2": 'two ',
        "3": 'three ',
        "4": 'four ',
        "5": 'five ',
        "6": 'six ',
        "7": 'seven ',
        "8": 'eight ',
        "9": 'nine '
    }

word_num = ''
for indy, i in enumerate(phone):
    word_num += p_num.get(phone[indy], "!")

print(word_num)

# ................................................................................................ #
# Emoji Converter:

message = input("Type a message > ")
words = message.split(" ")

emojis = {
    ":)": "    ",
    ":(": "    ",
    ":|": "    ",
    ":/": "    ",
    ":_": "    ",
    ";)": "    ",
    ">(": "    ",
    "XD": "    "
}
message_back = ''
for word in words:
    # If the word that the user types (the emoji, but in characters)
    # is in our emoji dictionary and has a corresponding emoji face
    # for that word, we will return it here, otherwise, we will use
    # whatever characters the user typed, i.e. the word.
    # The first word here supplies the key to the dictionary, and the
    # second tells the program what to put if match not found in keys.
    message_back += emojis.get(word, word) + " "
print(message_back)


# ................................................................................................ #
# FUNCTIONS: a reusable container for a few lines of code that perform a specific task
# Create a function to greet a user.
# When we write a line that ends in a colon, it means that what follows will be a block
# of code that belongs to the part before the colon.
# Code that is indented inside a function will ONLY execute when the function is called.

def greet_user():
    print("Hi there!")
    print("Welcome aboard!")


print("Start")
greet_user()
```

```python
print("Finish")

# ........................................................................................ #
# PARAMETERS: How to pass information to functions

def greet_user2(first_name, last_name):
    print(f"Hi {first_name} {last_name}!")
    print("Welcome aboard!")


print("Start")
greet_user2("John", "Smith")     # <- Here, we are passing the NAME John to our function as a parameter
greet_user2("Lola", "Johnson")     # <- This time, we call the function but with the variable Lola
print("Finish")

# When a function has a parameter, we are obligated to pass a value for that parameter.
# If we try to run the above function calls with no name in the parentheses as a
# parameter, we will get an TypeError message that we needed to pass a parameter but
# did not.
# PARAMETER: The holes or placeholders we define in our function for receiving information.
# ARGUMENT: The actual information that we supply to the function, to the hole.


# ........................................................................................ #
# POSITIONAL ARGUMENTS: Above, the first name and last name arguments passed to the function
# are positional arguments, meaning their position matters. The spot that they are when passed
# to the function will correlate to the parameters as passed to the function in its definition.

# KEYWORD ARGUMENTS: position does not matter for these. If we

def greet_again(first, last):
    print(f"Howdy, {first} {last}!")

# Here I have set the parameter FIRST to be John. Now, the position does not matter.
# The names have now been turned into a keyword argument.
greet_again(last = "Smith", first = "John")

# Most of the time, we will use positional arguments, but sometimes keyword arguments
# make code more READABLE, for example when using numbers, when it can be confusing
# as to which parameter different integers are targeting.

# Keyword arguments must always come AFTER positional arguments, if mixing the two.


# ........................................................................................ #
# Return Statement: Functions that return values

def square(to_square):
```

```
153     return to_square ** 2
154 
155 print(square(3))
156 
157 # If we merely print the result inside of the function rather than returning it and then
158 # printing, the Python interpreter will return None. By default, functions return None unless
159 # they are given a return statement telling them what to return. So None would be passed to
    the
160 # print statement on line 155.
161 
162 # .............................................................................................. #
163 # 08-21-22 Creating a Reusable Function (https://youtu.be/_uQrJ0TkZlc @ 2:49:07)
164 
165 # Turn the previous emoji project into its own function
166 
167 def emojify(message2):
168     words = message2.split(" ")
169 
170     emojis = {
171         ":)": "    ",
172         ":(": "    ",
173         ":|": "    ",
174         ":/": "    ",
175         ":_": "    ",
176         ";)": "    ",
177         ">(": "    ",
178         "XD": "    "
179     }
180     message_back2 = "
181     for word in words:
182         message_back2 += emojis.get(word, word) + " "
183 
184     return message_back2
185 
186 message2 = input("Type a message > ")
187 print("Emojified Message = ", emojify(message2))
188 
189 
```

```
 1  # ...................................................................................... #
 2  # 08-21-22 EXCEPTIONS / TRY EXCEPT (https://youtu.be/_uQrJ0TkZlc @ 2:53:54)
 3  # An exception is an error that crashes a program
 4  # How to handle errors
 5
 6  age = int(input("Age: "))
 7  print(age)
 8
 9  # We have told the program that the age input must be an int, but if a user inputs anything
10  # other than an int, they will get a ValueError code.
11
12  # TRY EXCEPT: used to avoid errors
13  # On except, add the kind of error that is most likely with this particular bit of code.
14
15  # This tells the program that if it runs into an error during the try block code
16  # that is of type ValueError, instead of giving the user the ValueError text, instead
17  # give them the printed message input in the except block.
18
19  # In the code below, we could also get a ZeroDivisionError, if the user inputs 0
20  # for their age and the program tries to use that for the line 18 operation.
21
22  try:
23      age_try = int(input("Age: "))
24      income = 20000
25      risk = income / age_try
26      print(age_try)
27
28  except ZeroDivisionError:
29      print("Age cannot be 0.")
30
31  except ValueError:
32      print("Invalid Value. Age must be a number.")
33
34
35  # ...................................................................................... #
36  # COMMENTS: Do not use comments to tell WHAT your code does, but rather whys and hows
    or other
37  # information that other developers would need to know about your code.
38  # Otherwise, verbose comments make code messy and redundant.
39
40  # ...................................................................................... #
41  # CLASSES: used to define new types of information
42  # Basic classes in Python include: numbers, strings, booleans
43  # Complex types of classes: Lists, dictionaries
44  # Use classes to define new types that model real concepts.
45
46  # New type: Point - with concepts and operations to work with and perform on points.
47  # Naming classes - capitalize the first letter of every word (Pascal Naming Convention)
48
49  class Point:
50      # Within the class - Define all the methods that belong to the class
51      def move(self):
```

```python
        print("move")

    def draw(self):
        print("draw")


# CLASS = defines the blueprint or the template for creating objects
# OBJECT = instance of a class based on the blueprint
# To create a new object, call on the class

point1 = Point()
# ATTRIBUTES: variables that belong to a particular object.
point1.x = 10
point1.y = 20

point1.draw()

print(point1.x)

point2 = Point()
point2.x = 1
print(point2.x)


# ........................................................................................................ #
# CONSTRUCTORS: A function that gets called at the time of creating an object of a class.
# In the above example, within our class Point, we did not originally create
# the attributes of x and y, which would always be an attribute of any point in space.
# Our constructor for our class attributes is the _init_ function

class Point2:
    # Defining the constructor, _init_, we pass it the parameters that it will be using whenever
    # it is called.
    def _init_(self, x, y):
        # To initialize our object with these parameters, we have to initialize each parameter.
        # Self here is a reference to the current object, then the argument follow is how we
        # set up and initialize each for each object we create. -> self.attribute = argument
        # This sets the value on the right of the = to be the attribute value for the object
        # we are creating

        self.x = x
        self.y = y

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

    def draw(self):
        print("draw")


# Now that we have created our constructor which includes how to assign x and y to each object
# of the class that is created, we can create an object in that class and easily assign those
# values by passing them upon creation.

point3 = Point2(10, 20)
print(point3.x)

# We can also change these values later:
point3.x = 11
print(point3.x)


# CHALLENGE: Create a person class with a name attribute and a talk() method

class Person:
    def _init_(self, name):
        self.name = name

    def talk(self):
        # person.name will return the name attribute of the current object.
        print(f"Hi, I am {self.name}!")


john = Person("John Smith")
john.talk()
bob = Person("Bob Bluebie")
bob.talk()


# ............................................................................................. #
# INHERITANCE: A mechanism for reusing code.
# Suppose we have a class Dog that has the method walk, but we also want to create a class Cat
# that will also have the method walk. We would have to repeat all the same code for defining
# the walk method under each class: dog, cat, and whatever other mammal we create.
# DO NOT REPEAT YOURSELF - DRY - cuz they all like to talk about this.
# If we create the same method inside of multiple classes, then if there ever needs to be
# a change to that method, it must be changed in every class where it appears, but...

# If we use INHERITANCE, we can create a parent class with multiple children that can
# inherit methods and attributes from the parent class.

class Mammal:
    def walk(self):
        print("Walk, animal!")


class Dog(Mammal):  # <- This is how we assign a class to a parent class so it can inherit
    def bark(self): # <- We can also define methods specific to child classes
        print('I am a dog. I go WOOF!')


class Cat(Mammal):
    def be_annoying(self):
        print("I am a cat. I am being annoying!")

dog1 = Dog()
dog1.walk()
dog1.bark()

cat1 = Cat()
cat1.be_annoying()

# ................................................................................ #
```

```python
# ........................................................................................ #
# 08-21-22 MODULES (https://youtu.be/_uQrJ0TkZlc @ 3:19:48)
# Modules are files with Python code. We use them to organize our code into multiple files,
# just like the fruits, vegetables, and other sections in a supermarket.

# We refer to each file as a module (REFER TO CONVERTER_MODULES.PY for this section)
# As we write more functions and classes, we put them in the module(file) to which they belong,
# organizing each by the fact that they relate to each other or work together.

import utils
# The module file is an object, so we can use . operator to access its members and call
# those functions:
print(utils.lbs_to_kgs(70))     # PRINTS: 31.5

# Instead of importing the entire module, you can also import a single function from inside
# the module:

from utils import lbs_to_kgs
# By importing this way, we do not have to prefix with the module(file) name.

print(lbs_to_kgs(30))   # PRINTS: 13.5

# CHALLENGE: create a function in utils.py called find_max that takes a list and
# returns the largest number from the list.
from utils import find_max
list01 = [4, 6, 12, 67, 3, 44, 13, 55]
print(find_max(list01))

# ........................................................................................ #
# PACKAGES: Big projects can contain hundreds or thousands of modules or files. Related modules
# can be organized inside of packages, containers for multiple files, a directory or folder.

# Imagine it like a department store that has different sections: Men's, Women's, Children's,
# and within those sections are subsections for various clothing types, etc.
# The Men's section could be seen as a package, and the outerwear section within the Men's
# section could be seen as a module.

# To create a package:
# 1) Create a new directory inside of your project with a descriptive name for the package.
# 2) Add a Python file called _init_ to the package directory. (Tells Python this is a package.)
# The files for this demonstration are in directory: package, module: shipping

# To import modules from a package, you have to give name of package . name of module.

import package.shipping

# This is a lot to type.
package.shipping.calc_shipping()

# So we can import like this:
from package.shipping import calc_shipping as ship
```

```python
ship()

# Or you can import the entire module, shorten its name and use the dot operator for individual
# functions within the module such as shipping.calc_shipping

# ........................................................................................... #
# PYTHON'S BUILT-IN MODULES: Python's documentation lists and explains all the built-in
# modules available. Mosh is gonna talk about the module random

import random
for i in range(3):
    print(random.random())     # <- by default, generates a random number between 0 and 1

# PRINTS:
# 0.8004578459479664
# 0.057541098205425745
# 0.7747335751454704

for i in range(3):
    print(random.randint(10, 20))   # <- randint gives integers, and the arguments are the range

team = ['John', 'Heather', 'Mary', 'Bob', 'Bill', 'Scooter', 'Bart', 'Chester', 'Louise']

leader = random.choice(team)    # <- Makes a random choice from a list passed to it.
print(leader)

# CHALLENGE: Make a dice roll program
# Define a class called DICE with a method called ROLL that returns a tuple with two random ints.


class Dice:
    def roll(self):
        x = random.randint(1, 6)
        y = random.randint(1, 6)
        roll_result = (x, y)

        return roll_result


dice = Dice()
print(dice.roll())

# ........................................................................................... #
# FILES and DIRECTORIES:
# Pathlib - library you can use to create objects to work with directories and files.
from pathlib import Path

# Absolute Path = start from root of hard disk to the directory going to the file
# Relative Path = starting from the current directory going to the file

```

```python
path = Path("package")
print(path.exists())        # <- Checks that the path is correct and exists.

# If you have set path = Path() and left out any other diretory, it will assume you mean
# the one you are in.

# MKDIR = to make a new directory (whatever directory path has been set to)
# RMDIR = to delete a directory (whatever directory path has been set to)

# GLOB = If you want to open all the files and directories in a path. Very useful if
# you are working on a program that opens a lot of files from other places.

# To get all the files (but  not directories) in a given path/directory, use path.glob("*.*)
# To get files of a certain type: path.glob('*.py') (for example, Python files). You will
# get a generator object, which you can loop through

for file in path.glob('*.py'):
    print(file)

# ........................................................................................ #
# PyPi and Pip: Python Package Index (Pip is how you open packages from PyPi)
# PyPi - tons of packages available. Some are not complete or contain bugs though

# WEB-SCRAPING - there are packages to help with this.
# Selenium - package for automating web app testing
```