This was CS50

Harvard College (https://www.college.harvard.edu/)
Fall 2021

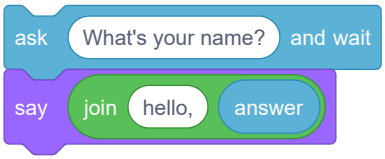
Lecture 6

Python syntax

- Today we'll learn a new programming language called Python, though our main goal is to learn how to teach ourselves new languages.
- Source code in Python looks a lot simpler than C, though it has many of the same ideas. To print "hello, world", all we need to write is:

```
print("hello, world")
```

  - Notice that, unlike in C, we don't need to specify a new line in the `print` function, or use a semicolon to end our line.
- In Scratch and C, we might have had multiple functions:



```
string answer = get_string("What's your name? ");
printf("hello, %s\n", answer);
```

- In Python, the equivalent would look like:

```
answer = get_string("What's your name? ")
print("hello, " + answer)
```

  - We can create a variable called `answer` without specifying the type, and we can join, or concatenate, two strings together with the `+` operator before we pass it into `print`.
  - The `get_string` function also comes from the Python version of the CS50 library.
- We can also write:

```
print(f"hello, {answer}")
```

  - The `f` before the double quotes indicates that this is a format string, which will allow us to use curly braces, `{}`, to include variables that should be substituted, or interpolated.
- We can create variables with just `counter = 0`. To increment a variable, we can write `counter = counter + 1` or `counter += 1`.
- Conditionals look like:

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

  - Unlike in C, where curly braces are used to indicate blocks of code, the exact indentation of each line determines the level of nesting in Python. And we don't need parentheses around the Boolean expressions.
  - And instead of `else if`, we just say `elif`.
- We can create a forever loop with a while loop:

```
while True:
    print("meow")
```

  - Both `True` and `False` are capitalized in Python.
- We can write a while loop with a variable:

```
i = 0
while i < 3:
    print("meow")
    i += 1
```

- We can write a `for` loop, where we can do something for each value in a list:

```
for i in [0, 1, 2]:
    print("hello, world")
```

  - Lists in Python, `[0, 1, 2]`, are like arrays or linked lists in C.
  - This `for` loop will set the variable `i` to `0`, run, then to the second value, `1`, run, and so on.
- And we can use a special function, `range`, to get any number of values:

```
for i in range(3):
    print("hello, world")
```

  - `range(3)` will give us a list up to but not including 3, with the values `0`, `1`, and `2`, that we can then use.
  - `range()` takes other arguments as well, so we can have lists that start at different values and have different increments between values.
- In Python, there are built-in data types similar to those in C:
  - `bool`, `True` or `False`
  - `float`, real numbers
  - `int`, integers which can grow as needed
  - `str`, strings
- Other types in Python include:
  - `range`, sequence of numbers
  - `list`, sequence of mutable values, or values we can change
  - `tuple`, sequence of immutable values
  - `dict`, dictionaries, collection of key/value pairs, like a hash table
  - `set`, collection of unique values, or values without duplicates
- The CS50 library for Python includes functions for getting user input as well:
  - `get_float`
  - `get_int`
  - `get_string`
- And we can import an entire library, functions one at a time, or multiple functions:

```
import cs50
```

```
from cs50 import get_float
from cs50 import get_int
from cs50 import get_string
```

```
from cs50 import get_float, get_int, get_string
```

```
from cs50 import get_float, get_int, get_string
```

Libraries

- Recall that, in C, we needed to compile a program with `make hello` before we could run it.
- To run a program we wrote in Python, we'll only need to run:

```
python hello.py
```

  - `python` is the name of a program called an **interpreter**, which reads in our source code and translates it to code that our CPU can understand, line by line.
  - Our source code files will also end in `.py`, to indicate that they're written in the Python language.
- For example, if our pseudocode for finding someone in a phone book was in Spanish and we didn't understand Spanish, we would have to slowly translate it, line by line, into English first:

```
1   Recoge guía telefónica
2   Abre a la mitad de guía telefónica
3   Ve la página
4   Si la persona está en la página
5       Llama a la persona
6   Si no, si la persona está antes de mitad de guía telefónica
7       Abre a la mitad de la mitad izquierda de la guía telefónica
8       Regresa a la línea 3
9   Si no, si la persona está después de mitad de guía telefónica
10      Abre a la mitad de la mitad derecha de la guía telefónica
11      Regresa a la línea 3
12  De lo contrario
13      Abandona
```

  - Similarly, programs in Python will take some extra time to be interpreted as they are run.
- We can blur an image with:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.BoxBlur(10))
after.save("out.bmp")
```

  - In Python, we include other libraries with `import`, and here we'll `import` the `Image` and `ImageFilter` names from the `PIL` library.
  - `Image` is an **object**, like a struct in C. Objects in Python can have not just values, but functions that we can access with the `.` syntax, such as with `Image.open`. And `before` is an object with a `filter` function as well, which we can find in the

`before` is an object with a `filter` function as well, which we can find in the documentation for the library.

- We can run this with `python blur.py` in the same directory as our `bridge.bmp` file:

```
filter/ $ ls
blur.py  bridge.bmp
filter/ $ python blur.py
filter/ $ ls
blur.py  bridge.bmp  out.bmp
filter/ $
```

- We can also find the edges in the image with:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.FIND_EDGES)
after.save("out.bmp")
```

- We can implement a dictionary with:

```
words = set()

def check(word):
    if word.lower() in words:
        return True
    else:
        return False

def load(dictionary):
  file = open(dictionary, "r")
  for line in file:
      word = line.rstrip()
      words.add(word)
  file.close()
  return True

def size():
    return len(words)

def unload():
    return True
```

  - First, we create a new set called `words`, which we can add values to, and have the language check for duplicates for us.
  - We'll define a function with `def`, and check if `word`, the argument, is in our hash table. (We'll also call a function `lower()` to get the lowercase version of the

table. (We'll also call a function, `.lower()`, to get the lowercase version of the word.)

- Our `load` function will take a file name, `dictionary`, and open it for reading. We'll iterate over the lines in the file with `for line in file:`, and add each word after removing each line's newline with `rstrip`.
- For `size`, we can use `len` to count the number of items in our dictionary, and finally, for `unload`, we don't have to do anything, since Python manages memory for us.

- We're able to run our program with `python speller.py texts/holmes.txt`, but we'll notice that it takes a few seconds longer to run than the C version. Even though it was much faster for us to write, we aren't able to fully optimize our code by way of managing memory and implementing all of the details ourselves.
- It turns out, we can cache, or save, the interpreted version of our Python program, so it runs faster after the first time. And Python is actually partially compiled too, into an intermediate step called bytecode, which is then run by the interpreter.

Input, conditions

- We can practice getting input from the user:

```
from cs50 import get_string

answer = get_string("What's your name? ")
print("hello, " + answer)
```

```
$ python hello.py
What's your name? David
hello, David
```

  - Notice that our program doesn't need a `main` function. Instead, our code will automatically run line by line.
  - We can also use a format string: `print(f"hello, {answer}")`.

- We can also use a function that comes with Python, `input`:

```
answer = input("What's your name? ")
print("hello, " + answer)
```

  - Since we're just getting a string from the user, `input` is the same as `get_string`.

- `get_int` and `get_float` will have error-checking for us, so we can get numeric values more easily:

```
from cs50 import get_int

x = get_int("x: ")
y = get_int("y: ")
print(x + y)
```

```
print(x + y)
```

- Since we're printing just one value, we can pass it to `print` directly.

- If we import the entire library, we see an error with a stack trace, or traceback:

```
import cs50

x = get_int("x: ")
y = get_int("y: ")
print(x + y)
```

```
$ python calculator.py
Traceback (most recent call last):
  File "/workspaces/20377622/calculator.py", line 3, in <module>
    x = get_int("x: ")
NameError: name 'get_int' is not defined
```

- It turns out that we need to write `cs50.get_int(...)` when we import the entire library. This allows us to **namespace** functions, or keep their names in different spaces, with different prefixes. Then, multiple libraries with a `get_int` function won't collide.

- If we call input ourselves, we get back strings for our values:

```
x = input("x: ")
y = input("y: ")
print(x + y)
```

```
$ python calculator.py
x: 1
y: 2
12
```

- And we print the two strings, joined together as another string.

- So we need to **cast**, or convert, each value from `input` into an `int` before we store it:

```
x = int(input("x: "))
y = int(input("y: "))
print(x + y)
```

- Notice that `int` in Python is a function that we can pass a value into.
- But if the user didn't type in a number, we'll need to do error-checking or our program will crash:

```
$ python calculator.py
x: cat
Traceback (most recent call last):
  File "/workspaces/20377622/calculator.py", line 1, in <module>
    x = int(input("x: "))
ValueError: invalid literal for int() with base 10: 'cat'
```

- `ValueError` is a type of **exception**, or something that goes wrong when our code is running. In Python, we can try to do something, and detect if there is an exception:

```
try:
    x = int(input("x: "))
except ValueError:
    print("That is not an int!)
    exit()
try:
    y = int(input("y: "))
except ValueError:
    print("That is not an int!)
    exit()
print(x + y)
```

  - Now, if there's an exception with converting the input into an integer, we can print a message and exit without crashing.

- We can divide values:

```
from cs50 import get_int

x = get_int("x: ")
y = get_int("y: ")

z = x / y
print(z)
```

```
$ python calculator.py
x: 1
y: 10
0.1
```

  - Notice that we get floating-point, decimal values back, even if we divided two integers. The division operator in Python doesn't truncate those values by default. (We can get the same behavior as in C, truncation, with the `//` operator, like `z = x // y`.)

- We can use a format string to print out more digits after the decimal point:

```
from cs50 import get_int

x = get_int("x: ")
y = get_int("y: ")

z = x / y
print(f"{z:.50f}")
```

```
$ python calculator.py
```

```
x: 1
y: 10
0.1000000000000000055511151231257827021181583404541 0
```

- Unfortunately, we still have floating-point imprecision.

- Comments in Python start with a `#`:

```
from cs50 import get_int

# Prompt user for points
points = get_int("How many points did you lose? ")

# Compare points against mine
if points < 2:
    print("You lost fewer points than me.")
elif points > 2:
    print("You lost more points than me.")
else:
    print("You lost the same number of points as me.")
```

- Our code is also much shorter than the same program in C.

- We can check the parity of a number with the remainder operator, `%`:

```
from cs50 import get_int

n = get_int("n: ")

if n % 2 == 0:
    print("even")
else:
    print("odd")
```

```
$ python parity.py
n: 50
even
```

- To compare strings, we can say:

```
from cs50 import get_string

s = get_string("Do you agree? ")

if s == "Y" or s == "y":
    print("Agreed.")
elif s == "N" or s == "n":
    print("Not agreed.")
```

```
$ python agree.py
Do you agree? y
```

```
Agreed.
```

- Python doesn't have a data type for single characters, so we check `Y` and other letters as strings. (We can use either single or double quotes for strings, too, as long as we're consistent.)
- We can compare strings directly with `==`, and we can use `or` and `and` in our Boolean expressions.

- We can also check if our string is in a list, after converting it to lowercase first:

```
from cs50 import get_string

s = get_string("Do you agree? ")

s = s.lower()

if s in ["y", "yes"]:
    print("Agreed.")
elif s in ["n", "no"]:
    print("Not agreed.")
```

- We call `s.lower()` to get the lowercase version of the string, and then store it back in `s`.
- We can even just say `s = get_string("Do you agree? ").lower()` to convert the input to lowercase immediately, before we store it in `s`.

- In Python, strings are also immutable, or unchangeable. When we make changes to a string, a new copy is made for us, along with all the memory management.

meow

- We can demonstrate design improvements to our `meow` program, too:

```
print("meow")
print("meow")
print("meow")
```

- Here, we have the same line of code three times.

- We can use a loop:

```
for i in range(3):
    print("meow")
```

- We can define a function that we can reuse:

```
for i in range(3):
    meow()

def meow():
    print("meow")
```

- But this causes an error when we try to run it: `NameError: name 'meow' is not defined`. It turns out that, like in C, we need to define our function before we use it.

  So we'll define a `main` function first:

  ```
  def main():
      for i in range(3):
          meow()

  def meow():
      print("meow")

  main()
  ```

  - Now, by the time we actually call our `main` function at the end of our program, the `meow` function will already have been defined.
  - The important part of our code will still be at the top of our file, so it's easy to find.
- We might also see examples that call a `main` function with:

  ```
  if __name__ == "__main__":
      main()
  ```

  - This solves problems with including our code in libraries, but we won't need to consider that yet, so we can simply call `main()`.
- Our functions can take arguments, too:

  ```
  def main():
      meow(3)

  def meow(n):
      for i in range(n):
          print("meow")

  main()
  ```

  - Our `meow` function takes in a parameter, `n`, and passes it to `range` in a for loop.
  - Notice that we don't need to specify the type of an argument.
- We can create global variables by initializing them outside of `main`, though Python doesn't have constants.

Mario

- We can print out a row of hash symbols on the screen:

  ```
  from cs50 import get_int
  ```

```
n = get_int("Height: ")

for i in range(n):
    print("#")
```

```
$ python mario.py
Height: -1
```

- If the user passes in a negative number, we see no output. Instead, we should prompt the user again.

- In Python, there is no do while loop, but we can achieve the same effect:

```
from cs50 import get_int

while True:
    n = get_int("Height: ")
    if n > 0:
        break

for i in range(n):
    print("#")
```

- We'll write an infinite loop, so we do something at least once, and then use `break` to exit the loop if we've met some condition.

- We can use a helper function:

```
from cs50 import get_int

def main():
    height = get_height()
    for i in range(height):
        print("#")

def get_height():
    while True:
        n = get_int("Height: ")
        if n > 0:
            break
    return n

main()
```

- Our `get_height()` function will return `n` after it meets our condition. Notice that, in Python, variables are scoped to a function, meaning we can use them outside of the loop they're created in.

- We can use `input` ourselves:

```
def main():
    height = get_height()
```

```
    for i in range(height):
        print("#")

def get_height():
    while True:
        n = int(input("Height: "))
        if n > 0:
            break
    return n

main()
```

```
$ python mario.py
Height: cat
Traceback (most recent call last):
  File "/workspaces/20377622/mario.py", line 13, in <module>
    main()
  File "/workspaces/20377622/mario.py", line 2, in main
    height = get_height()
  File "/workspaces/20377622/mario.py", line 8, in get_height
    n = int(input("Height: "))
ValueError: invalid literal for int() with base 10: 'cat'
```

- If we don't get a valid input, our traceback shows the stack of functions that led to the exception.

- We'll `try` to convert the input to an integer, and `print` a message if there is an exception. But we don't need to exit, since our loop will prompt the user again. If there isn't an exception, we'll continue to check if the value is positive and `break` if so:

```
def main():
    height = get_height()
    for i in range(height):
        print("#")

def get_height():
    while True:
        try:
            n = int(input("Height: "))
            if n > 0:
                break
        except ValueError:
            print("That's not an integer!")
    return n

main()
```

- Now we can try to print question marks on the same line:

```
for i in range(4):
```

```
    print("?", end="")
print()
```

```
$ python mario.py
????
```

- When we print each question mark, we don't want the automatic new line, so we can pass a **named argument**, also known as keyword argument, to the `print` function. (So far, we've only seen **positional arguments**, where arguments are set based on their position in the function call.)
- Here, we pass in `end=""` to specify that nothing should be printed at the end of our string. If we look at the documentation for `print`, we'll see that the default value for `end` is `\n`, a new line.
- Finally, after we print our row with the loop, we can call `print` with no other arguments to get a new line.

- We can also use the multiply operator to join a string to itself many times, and print that directly with: `print("?" * 4)`.
- We can implement nested loops, too:

```
for i in range(3):
    for j in range(3):
        print("#", end="")
    print()
```

```
$ python mario.py
###
###
###
```

Documentation

- The official Python documentation (https://docs.python.org/) includes references for built-in functions.
- We can use the search function to find a page about functions that come with strings, for example, including the `lower()` (https://docs.python.org/3/library/stdtypes.html?highlight=lower#str.lower) function for converting a string to lowercase. On the same page, we'll see lots of other functions, though we shouldn't worry about learning all of them immediately.

Lists, strings

- We can create a list:

```
scores = [72, 73, 33]



average = sum(scores) / len(scores)
print(f"Average: {average}")
```

- We can use `sum`, a function built into Python, to add up the values in our list, and divide it by the number of scores, using the `len` function to get the length of the list.

- We can add items to a list with:

```
from cs50 import get_int

scores = []
for i in range(3):
    score = get_int("Score: ")
    scores.append(score)

average = sum(scores) / len(scores)
print(f"Average: {average}")
```

  - With the `append` method, a function built into list objects, we can add new values to `scores`.
  - We can also join two lists with `scores += [score]`. Notice that we need to put `score` into a list of its own.

- We can iterate over each character in a string:

```
from cs50 import get_string

before = get_string("Before:  ")
print("After:  ", end="")
for c in before:
    print(c.upper(), end="")
print()
```

  - Python will iterate over each character in the string for us with just `for c in before:`.

- To make a string uppercase, we can also just write `after = before.upper()`, without having to iterate over each character ourselves.

Command-line arguments, exit codes

- We can take command-line arguments with:

```
from sys import argv
```

```
if len(argv) == 2:
    print(f"hello, {argv[1]}")
else:
    print("hello, world")
```

```
$ python argv.py
hello, world
$ python argv.py David
hello, David
```

- We import `argv` from `sys`, the system module, built into Python.
- Since `argv` is a list, we can get the second item with `argv[1]`.
- `argv[0]` would be the name of our program, like `argv.py`, and not `python`.

- We can also let Python iterate over the list for us:

```
from sys import argv

for arg in argv:
    print(arg)
```

```
$ python argv.py
argv.py
$ python argv.py foo bar baz
argv.py
foo
bar
baz
```

- With Python, we can start at a different index in a list:

```
for arg in argv[1:]:
    print(arg)
```

- This lets us **slice** the list from 1 to the end.
- We can write `argv[:-1]` to get everything in the list except the last element.

- We can return exit codes when our program exits, too:

```
from sys import argv, exit

if len(argv) != 2:
    print("Missing command-line argument")
    exit(1)

print(f"hello, {argv[1]}")
exit(0)
```

- Now, we can use `exit()` to exit our program with a specific code.

- We can import the entire `sys` library, and make it clear in our program where these

functions come from:

```
import sys

if len(sys.argv) != 2:
    print("Missing command-line argument")
    sys.exit(1)

print(f"hello, {sys.argv[1]}")
sys.exit(0)
```

```
$ python exit.py
Missing command-line argument
$ python exit.py David
hello, David
```

Algorithms

- We can implement linear search by checking each element in a list:

```
import sys

numbers = [4, 6, 8, 2, 7, 5, 0]

if 0 in numbers:
    print("Found")
    sys.exit(0)

print("Not found")
sys.exit(1)
```

  - With `if 0 in numbers:`, we're asking Python to check the list for us.
- A list of strings, too, can be searched with:

```
import sys

names = ["Bill", "Charlie", "Fred", "George", "Ginny", "Percy", "Ron"]

if "Ron" in names:
    print("Found")
    sys.exit(0)

print("Not found")
sys.exit(1)
```

- If we have a dictionary, a set of key-value pairs, we can also check for a particular key, and look at the value stored for it:

```
from cs50 import get_string

people = {
    "Carter": "+1-617-495-1000",
    "David": "+1-949-468-2750"
}

name = get_string("Name: ")
if name in people:
    number = people[name]
    print(f"Number: {number}")
```

```
$ python phonebook.py
Name: David
Number: +1-949-468-2750
```

- We first declare a dictionary, `people`, where the keys are strings of each name we want to store, and the value for each key is a string of a corresponding phone number.
- Then, we use `if name in people:` to search the keys of our dictionary for a `name`. If the key exists, then we can get the value with the bracket notation, `people[name]`.

Files

- Let's open a CSV file, with comma-separated values:

```
import csv
from cs50 import get_string

file = open("phonebook.csv", "a")

name = get_string("Name: ")
number = get_string("Number: ")

writer = csv.writer(file)
writer.writerow([name, number])

file.close()
```

```
$ python phonebook.py
Name: Carter
Number: +1-617-495-1000
$ ls
phonebook.csv  phonebook.py
```

- It turns out that Python also has a `csv` library that helps us work with CSV files, so after we open the file for appending, we can call `writer` to create a `writer`

after we open the file for appending, we can call `csv.writer` to create a `writer` object from the file. Then, we can use a method inside it, `writer.writerow`, to write a list as a row.

- Our `phonebook.csv` file will have our data:

```
Carter,+1-617-495-1000
```

  - We can run our program again, and see new data being added to our file.
- We can use the `with` keyword, which will close the file for us after we're finished:

```
...
with open("phonebook.csv", "a") as file:
    writer = csv.writer(file)
    writer.writerow((name, number))
```

- We'll visit a Google Form, and select a "house" (https://harrypotter.fandom.com/wiki/Hogwarts_Houses) we might want to be in.
- We'll download the data as a CSV file, which looks like this:

```
  Timestamp,House
  10/13/2021 16:00:07,Ravenclaw
  10/13/2021 16:00:07,Gryffindor
  10/13/2021 16:00:09,Ravenclaw
  10/13/2021 16:00:10,Gryffindor
  10/13/2021 16:00:10,Gryffindor
  ...
```

- Now we can tally the number of times a house appears:

```
import csv

houses = {
    "Gryffindor": 0,
    "Hufflepuff": 0,
    "Ravenclaw": 0,
    "Slytherin": 0
}

with open("hogwarts.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        house = row[1]
        houses[house] += 1

for house in houses:
    count = houses[house]
    print(f"{house}: {count}")
```

    - We use the `reader` function from the `csv` library, skip the header row with `next(reader)`, and then iterate over each of the rest of the rows.

    - The second item in each row, `row[1]`, is the string of a house, so we can use that to access the value stored in `houses` for that key, and add one to it with `houses[house] += 1`.
    - Finally, we'll print out the count for each house.
- We can improve our program by reading each row as a dictionary, using the first row in the file as the keys for each value:

```
...
with open("hogwarts.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        house = row["House"]
        houses[house] += 1
...
```

    - Now, we can say `house = row["House"]` to get the value in that column.

More libraries

- On our own Mac or PC, we can use another library to convert text to speech (since VS Code in the cloud doesn't support audio):

```
import pyttsx3

engine = pyttsx3.init()
engine.say("hello, world")
engine.runAndWait()
```

    - By reading the documentation, we can use a Python library called `pyttsx3` to play some string as audio.
    - We can even pass in a format string with `engine.say(f"hello, {name}")` to say some input.
- We can use another library, `face_recognition`, to find faces in images with `detect.py` (https://cdn.cs50.net/2021/fall/lectures/6/src6/6/faces/detect.py?highlight):

```
# Find faces in picture
# https://github.com/ageitgey/face_recognition/blob/master/examples/find_faces

from PIL import Image
import face_recognition

# Load the jpg file into a numpy array
image = face_recognition.load_image_file("office.jpg")
```

```
# Find all the faces in the image using the default HOG-based model.
# This method is fairly accurate, but not as accurate as the CNN model and not

# See also: find_faces_in_picture_cnn.py
face_locations = face_recognition.face_locations(image)

for face_location in face_locations:

    # Print the location of each face in this image
    top, right, bottom, left = face_location

    # You can access the actual face itself like this:
    face_image = image[top:bottom, left:right]
    pil_image = Image.fromarray(face_image)
    pil_image.show()
```

- In `recognize.py` (https://cdn.cs50.net/2020/fall/lectures/6/src6/6/faces/recognize.py), we can see a program that finds a match for a particular face.
- In `listen0.py` (https://cdn.cs50.net/2021/fall/lectures/6/src6/6/listen/listen0.py?highlight), we can respond to input from the user:

```
# Recognizes a greeting

# Get input
words = input("Say something!\n").lower()

# Respond to speech
if "hello" in words:
    print("Hello to you too!")
elif "how are you" in words:
    print("I am well, thanks!")
elif "goodbye" in words:
    print("Goodbye to you too!")
else:
    print("Huh?")
```

- We can recognize audio input from a microphone and respond with `listen2.py` (https://cdn.cs50.net/2021/fall/lectures/6/src6/6/listen/listen2.py?highlight):

```
# Responds to a greeting
# https://pypi.org/project/SpeechRecognition/

import speech_recognition

# Obtain audio from the microphone
recognizer = speech_recognition.Recognizer()
with speech_recognition.Microphone() as source:
    print("Say something:")
```

```
    audio = recognizer.listen(source)

# Recognize speech using Google Speech Recognition

words = recognizer.recognize_google(audio)

# Respond to speech
if "hello" in words:
    print("Hello to you too!")
elif "how are you" in words:
    print("I am well, thanks!")
elif "goodbye" in words:
    print("Goodbye to you too!")
else:
    print("Huh?")
```

- We can even add more logic to listen for a name:

```
# Responds to a name
# https://pypi.org/project/SpeechRecognition/

import re
import speech_recognition

# Obtain audio from the microphone
recognizer = speech_recognition.Recognizer()
with speech_recognition.Microphone() as source:
    print("Say something:")
    audio = recognizer.listen(source)

# Recognize speech using Google Speech Recognition
words = recognizer.recognize_google(audio)

# Respond to speech
matches = re.search("my name is (.*)", words)
if matches:
    print(f"Hey, {matches[1]}.")
else:
    print("Hey, you.")
```

- We can create a QR code (https://en.wikipedia.org/wiki/QR_code), or two-dimensional barcode, with another library:

```
import os
import qrcode

img = qrcode.make("https://youtu.be/xvFZjo5PgG0")
img.save("qr.png", "PNG")
os.system("open qr.png")
```

- Now, when we run our program, a QR code will be generated and opened.