# Day 5

This is CS50 for MBAs. Harvard Busines School. Spring 2015.

Cheng Gong

## Table of Contents

Questions ... 1
Vault-Hide ... 2
Python ... 8

## Questions

- Why would it not be accurate to describe HTML as a programming language?

  # HTML tells us how to structure something, but there's no real logic like looping or conditions. It does, however, provide us with an interface that we'll be able to manipulate with JavaScript, which *is* a programming language. And today we'll look at a back-end, which is what servers use to generate HTML, in a language called Python.

- What does CSS allow you to do that HTML alone does not?

  # CSS allows you to stylize our webpage and separate the presentation from structure. It also standardizes the appearance of our pages since we can control it more precisely.

- Ending times are now on the syllabus.[1]

- If I do 'view page source' of CS50 for MBAs website, I see html code that I can more or less understand now. Why is the content of Google home page's 'view page source' SO long and complex? What is included in the code?

  # The CS50 website is static, meaning there's nothing that really changes after we save something to it. Google's homepage has the search box that starts changing as soon as you type into it, with suggestions that drop down based on what you type

[1] http://cs50.net/mba/syllabus/

in, not to mention the other features it supports, so there's more code involved. It also minified, or compressed the code, like we talked about last time.

- I was unable to access the CS50 website again this afternoon. I received the same message from my Chrome browser as I did on the first day, saying the DNS server had an issue. Luckily, I had just learned how to manually change my DNS server to Google's public version, and I did so and am now happy be using the site to send this message. I assume this was a test, Professor Malan…very clever.

# This wasn't a test. There might actually be a DNS server on HBS' campus that hasn't yet been updated, so if you have this problem, tell us.

- I'm trying to get my new web page to search the weather web page. I can get to a page where you type in a zipcode to the box, but when I press the button, nothing happens. My code looks as below. Any sense of what is wrong?

```
<body>
    ....
    <form actions="http://www.wunderground.com" method="get">
        <input name="q" type="text"/>
        <br/>
        <input type="submit" value="Let's see"/>
    </form>
</body>
```

# `actions` should be `action`, and sometimes all it takes is one simple typo to break everything. But there are now tools, IDEs, that help catch issues like this.

# Multiple input boxes on the same page are just multiple form elements, and to see how they are implemented, we can open Chrome's Developer Tools.

# The input name, `q`, can also be changed by the developer on the other side of the form.

## Vault-Hide

- An Android app called Vault-Hide is pretty popular and well-reviewed:

Vault-Hide SMS, Pics & Videos

NQ Mobile Security (NYSE:NQ) - April 3, 2015

Business

Install    Add to Wishlist

Offers in-app purchases

(467,035)

- The program’s main feature seems to be keeping your photos and video secure:

**Top features of Vault**

☆**Photos & videos protection:** All files will be stored in a safe place and can only be viewed in Vault after correct password.

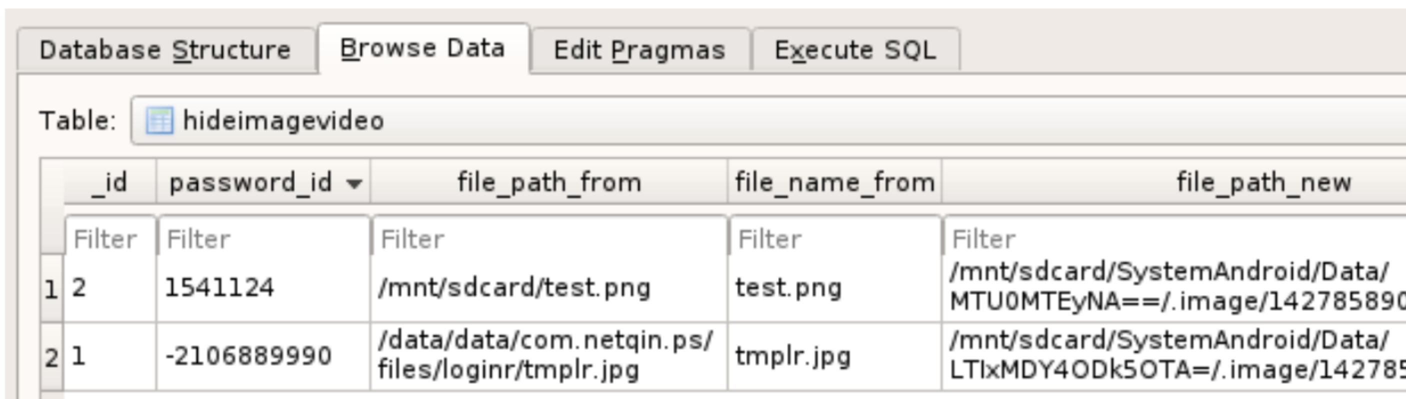
- There’s a blog post the other day, from someone who managed to break the "encryption" of the app. Though to be fair, the description never mentioned encryption, just that it would keep things "secure."
- So this person, ninjadoge24[2] created a small image as a test, displayed here as hexadecimal in the middle column:

[2] https://ninjadoge24.github.io/

```
0000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452  .PNG.....
0000010: 0000 0001 0000 0001 0802 0000 0090 7753  .........
0000020: de00 0000 0970 4859 7300 0003 b100 0003  .....pHYs.
0000030: b101 f583 ed49 0000 0007 7449 4d45 07df  .....I....
0000040: 0401 0319 3a3d ca0b 0c00 0000 0c69 5458  ....:=....
0000050: 7443 6f6d 6d65 6e74 0000 0000 00bc aeb2  tComment..
0000060: 9900 0000 0f49 4441 5408 1d01 0400 fbff  .....IDAT.
0000070: 00ff 0000 0301 0100 c706 926f 0000 0000  .........
0000080: 4945 4e44 ae42 6082 4e49 4e4a 4144 4f47  IEND.B`.NI
0000090: 4532 340a                                E24.
```

# The column on the left is the location of each row in bytes, and the column on the right is an ASCII representation of the data, where the `.` isn't a literal period but binary data that doesn't have a printable character.

- We see that our friend added his or her name to the end of the file, just so we can recognize it more easily.
- Then our friend looked into the database on the phone and noticed that `test.png` was actually stored in the file path there:



Database Structure | Browse Data | Edit Pragmas | Execute SQL

Table: hideimagevideo

| | _id | password_id | file_path_from | file_name_from | file_path_new |
|---|---|---|---|---|---|
| | Filter | Filter | Filter | Filter | Filter |
| 1 | 2 | 1541124 | /mnt/sdcard/test.png | test.png | /mnt/sdcard/SystemAndroid/Data/ MTU0MTEyNA==/.image/14278589 |
| 2 | 1 | -2106889990 | /data/data/com.netqin.ps/ files/loginr/tmplr.jpg | tmplr.jpg | /mnt/sdcard/SystemAndroid/Data/ LTIxMDY4ODk5OTA=/.image/142785 |

- That file was the same size as before, and a little different at the beginning, but it seems that some of the file hasn't been changed.

```
0000000: 8d54 4a43 090e 1e0e 0404 0409 4d4c 4056  .TJC.....
0000010: 0404 0405 0404 0405 0c06 0404 0494 7357  .........
0000020: da04 0404 0d74 4c5d 7704 0407 b504 0407  .....tL]w.
0000030: b505 f187 e94d 0404 0403 704d 4941 03db  .....M...
0000040: 0005 071d 3e39 ce0f 0804 0404 086d 505c  ....>9...
0000050: 7047 6b69 6961 6a70 0404 0404 04b8 aab6  pGkiiajp..
0000060: 9d04 0404 0b4d 4045 500c 1905 0004 fffb  .....M@EP.
0000070: 04fb 0404 0705 0504 c302 966b 0404 0404  .........
0000080: 4945 4e44 ae42 6082 4e49 4e4a 4144 4f47  IEND.B`.NI
0000090: 4532 340a                                E24.
```

- If we use the XOR operation to compare the two, it seems that the difference between the two versions of the file was just `04` over and over again:

```
89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49
                 XOR
8D 54 4A 43 09 0E 1E 0E 04 04 04 09 4D
                  =
04 04 04 04 04 04 04 04 04 04 04 04 04
```

  # Remember that XOR compares two bits, and hexadecimal represents 4 bits in each of its digits, so we can do this operation.

- Then our friend made a bigger image file, and found that only the first part of the file was only being changed, this time with a key of `cc`:

```
0000000: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000010: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000020: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000030: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000040: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000050: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000060: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000070: cccc cccc cccc cccc cccc cccc cccc cccc  .........
0000080: 0000 0000 0000 0000 0000 0000 0000 0000  .........
0000090: 0000 0000 0000 0000 0000 0000 0000 0000  .........
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  .........
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  .........
                    -- snip --
0000590: 0000 0000 0000 0000 0000 0000 0000 0000  .........
00005a0: 0000 0000 0000 0000 0000 0000 0000 0000  .........
00005b0: 0000 0000 0000 0000 0000 0000 0000 0000  .........
00005c0: 0000 0000 0000 0000 0000 0000 0000 0000  .........
00005d0: 0000 0000 0000 0000 0000 0000 0a         .........
```

- The Vault app also asks for a password, so through trial and error our friend noticed that there was some sort of pattern in the password and the key:

| password | key |
| --- | --- |
| 000 | 30 |
| 001 | 31 |
| 002 | 32 |
| 003 | 33 |
| 004 | 34 |
| ... | |
| 010 | 4f |
| 011 | 50 |
| 012 | 51 |
| ... | |
| 2424 | 04 |
| 4815162342 | cc |

- More importantly, though, there are only 8 bits in the key, so there are only 256 possible values (255 if we don't count 0).
- Instead of trying to figure out the key for every password, it was easier to just write some code that tried all the possible keys, since there are so few:

```
#!/bin/sh
for i in `seq 0 255`; do
    ./vault-crack $1 $i > $1.decrypted
    if [ `file $1.decrypted --brief --mime-type` != "application/octet-
    then
        echo "Key = $i" && exit
    fi
done
```

- So the implication now is that files "protected" by this app aren't all that secure.

## Python

- We're gonna write some code, not too different than what we did with those Scratch blocks, in a language called Python.
- Remember that blocks like `[ say ]` are statements, and blocks like `< mouse down? >` are boolean expressions.
- Blocks like `[ if < > ]` are conditions.
- Bill Gates[3] talks about conditions, and Mark Zuckerberg[4] talks about loops.
- Rememebr that some languages need to be compiled from source code to object code by a compiler, turning the words we typed into binary instructions that computers can understand.
- Interpreted languages are a little different, because we can skip the compilation process. This doesn't make much of a difference with smaller programs, but definitely does with larger ones.
- An interpreter is a program that reads source code in a particular language and converts it to instructions line by line as it is run.
- Let's start Python by converting our familiar:

  ```
  [ when green flag clicked ]
  [ say [hello, world] ]
  ```

  to this:

[3] https://www.youtube.com/watch?v=m2Ux2PnJe6E
[4] https://www.youtube.com/watch?v=mgooqyWMTxk

```
if __name__ == "__main__":
    print("hello, world")
```

# You can see that this is logically similar, even if the syntax is a bit different.
# The first line is the equivalent of the start of the program, sort of like a `main` function.

- The `[ say ]` block has been converted to `print`, with quotes around what we want to print.
- A `[ forever ]` loop in Python looks like this:

```
while True:
    print("hello, world")
```

# The indentation is how we place statements inside the loop, rather than outside.
# The `True` can be replaced by a boolean, and the `while` checks whatever the expression is and runs the statements if it's true. In this case, `True` is always true, so this loop will execute forever.

- A music player, clock, or spell-checker are examples of programs we might want to run continuously.
- A `while False:` condition would be like a `[ never ]` block in Scratch, since nothing inside could ever run.
- To disable code temporarily, we would "comment it out" with special syntax that tells the interpreter to ignore it.
- There's also case sensitivity here, where `True` is capitalized since it's a predefined word.
- We could replace `while True:` with `if 1 == 1:`, but that would only execute once.
- A `[ repeat [10] ]` block is written like this:

```
for i in range(10):
    print("hello, world")
```

- `for` starts a finite loop, where `i` is a variable, and `range(10)`, which is a function that generates a list of values that start with `0` and ends with `9`.
- So it's equivalent to this:

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
```

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

# But we don't want to do this, since we're more likely to make a mistake, and `range` is easier to read.

- We can also convert this:

```
[ set counter to 0 ]
[ forever ]
| [ say counter ]
| [ change counter by 1 ]
____
```

to this:

```
counter = 0
while True:
    print(counter)
    counter += 1
```

# The strange syntax of `+=` means we're adding the value on the right, in this case `1`, and saving back to the variable `counter`.

- Python also doesn't use braces to separate statements in a loop from those outside, but rather indentation.
- Booleans in Python are simply `x < y` or `(x < y) and (y < z)`, if we wanted to compare variables as a condition.
- For example, we could convert this:

```
[ if x < y  ]
| [ say x is less than y ]
[ else ]
| [ if x < y  ]
| | [ say x is greater than y ]
| [ else ]
| | [ say x is equal to y ]
| ____
____
```

to this:

```
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")
```

# `elif` means `else if`, which other languages use, but Python uses the word `elif`. Line by line, this piece of code translates to Scratch pretty straightforwardly.

- Python also has lists, which we've seen in Scratch like this:

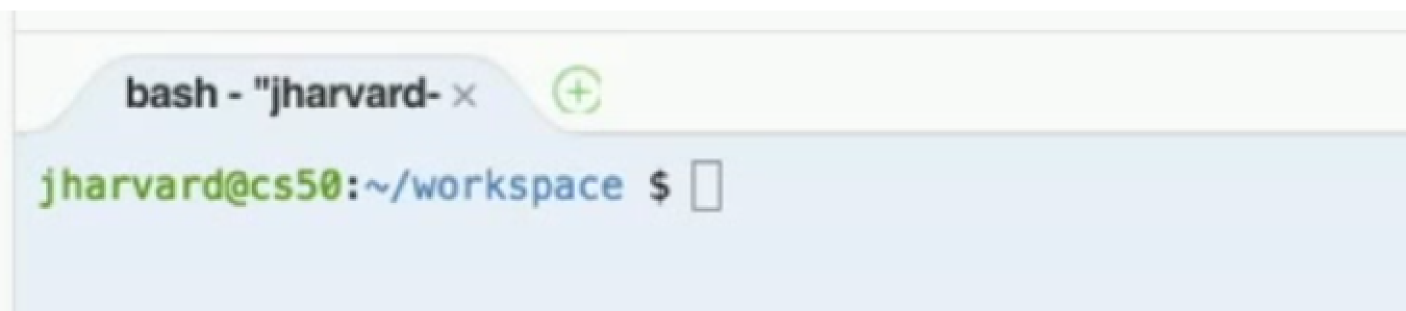
```
[ add [Orange] to [inventory] ]
```

- In Python, lists are like arrays in that we can access any element with the `[]` notation, but are flexible in size. (The cost of this, though, is speed and memory since under the hood the interpreter has to keep track of all the elements somehow. An interpreted language too, is slow, since it needs to be converted to binary as it's run, but in return we get more developer time since we don't need to wait for code to compile.)

```
inventory = []
inventory.append("Orange")
```

# To get "Orange" back, we'd use something like `inventory[0]`.

- We'll switch to our Cloud9 workspace for a demo, located at ide.c9.io/jharvard/cs50[5].
- The same source files are here[6].
- The bottom panel of the workspace looks like this:




[5] https://ide.c9.io/jharvard/cs50
[6] http://cdn.cs50.net/2015/mba/classes/5/src5/

    - `bash` is the name of the program currently running, which is like a command line that takes in what we type.
    - `jharvard` is our username, `cs50` is the name of the computer, and `~/workspace` is the name of the folder we're in.
- The contents of our `workspace` folder is in the panel to the left:

Workspace | Navigate | Commands

- cs50
  - src4
  - src5
    - adder.py
    - ages.py
    - conditions-0.py
    - conditions-1.py
    - conditions-2.py
    - f2c.py
    - floats-0.py
    - floats-1.py
    - floats-2.py
    - function-0.py
    - hello-0.py
    - hello-1.py
    - hello-2.py
    - hello-3
    - hello-4
    - positive.py
    - return.py
    - string-0.py
  - README.md

- We're gonna switch to our Mac, since our (free and thus small) Cloud9 workspace was overwhelmed by everyone in the class logging in at the same time.
- We'll use TextEdit to make a super simple file and save it as `hello.py`:

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

- Then in the Terminal, we'll see that it works:

```
% python hello.py
hello, world
%
```

  # (On David's computer, he's preconfigured his terminal to output just `%` instead of his username and directory, like Cloud9 did.)

- Next, we'll store the name of a person in a variable:

```
name = "David"
print("hello, name")
```

- But we don't get what we intended:

```
% python hello.py
hello, name
%
```

- So we need to do this:

```
name = "David"
print("hello, {}".format(name))
```

  # The `{}` is like a placeholder, and `name` is the variable that will go in that placeholder. `.format` is accessing the function of the string that formats it, which comes with Python for all strings, and in this case it takes `name` and puts it into `{}`.

  # The double quotes around `David` is to indicate that this is a string, rather than a reference to another variable or function.

- Now it works as expected:

```
% python hello.py
```

```
hello, David
%
```

- Instead of `David`, we can try this:

```
name = raw_input()
print("hello, {}".format(name))
```

  # `raw_input()` is a function (since it has the parentheses afterwards) that comes with Python that we can call, which will ask the user to type in a string when the program is run:

  ```
  % python hello.py
  Lin
  hello, Lin
  % python hello.py
  David
  hello, David
  %
  ```

  # The variable `name` stores whatever we typed in, and is printed back out again by `print`.

- Once we get familiar with the syntax, things become more straightforward, since we can look up functions or how to do things as we need them, and put them together like blocks in Scratch to accomplish what we need.

- If we want to print `{}` as brackets, we can say:

```
name = raw_input()
print("hello, {}{{}}".format(name))
```

- We're "escaping" the braces here by repeating them the second time (a Python convention we can look up), so it prints out something like `hello, David {}`.

- Let's open adder.py[7]:

[7] http://cdn.cs50.net/2015/mba/classes/5/src5/adder.py

```
"""Adds two numbers."""

x = int(raw_input("Give me an integer: "))
y = int(raw_input("Give me another integer: "))

print("The sum of {} and {} is {}!".format(x, y, x + y))
```

- The first line is a comment, which means the interpreter will ignore it, but it's useful for us since other people can read it.
- The lines with `x` and `y` seem to be getting input from the user again, but the string is being turned to an integer with the `int()` function. `x = int("123")`, for example, will turn the string `"123"` into the integer value `123`.
- Within the `raw_input` function, the string we pass in will be printed for the user before we wait for input, just to help the user know that input is expected. And we only know that we can do this because we looked it up in the documentation.
- Finally, we print the numbers `x` and `y` and their sum, using not one but now three placeholders `{}`, and passing in all three values to our `format` function, separating them with commas.
- Now we can run it:

```
% python adder.py
Give me an integer: 1
Give me another integer: 2
The sum of 1 and 2 is 3!
%
```

- Alternatively, if we end up with too many variables, we could use this format for `format` (pun intended):

```
"""Adds two numbers."""

x = int(raw_input("Give me an integer: "))
y = int(raw_input("Give me another integer: "))

print("The sum of {first} and {second} is {third}!".format({
    "first": x,
    "second": y,
    "third": x + y
}))
```

# Now we're giving names to each of the expressions to put them in the correct place.

- Let's look at conditions-0.py[8]:

```
"""Tells user if his or her input is positive or negative (somewhat
 inaccurately). Demonstrates use of if-else construct."""

# ask user for an integer
n = int(raw_input("I'd like an integer please: "))

# analyze user's input (somewhat inaccurately)
if n > 0:
    print("You picked a positive number!")
else:
    print("You picked a negative number!")
```

# The `#` symbol indicates a comment of one line, just like `"""`.

# This program also gets an integer from the user, but it's inaccurate because `0` will be categoried as negative.

- We can fix this in conditions-1.py[9]:

```
"""Tells user if his or her input is positive or negative. Demonstrates
 use of if-elif-else construct."""

# ask user for an integer
n = int(raw_input("I'd like an integer please: "))

# analyze user's input (somewhat inaccurately)
if n > 0:
    print("You picked a positive number!")
elif n == 0:
    print("You picked zero!")
else:
    print("You picked a negative number!")
```

# The `elif` means "else if", so now there are three cases.

- Equivalently, we could do this, since the order of our conditions doesn't matter:

[8] http://cdn.cs50.net/2015/mba/classes/5/src5/conditions-0.py
[9] http://cdn.cs50.net/2015/mba/classes/5/src5/conditions-1.py

```
"""Tells user if his or her input is positive or negative. Demonstrates
 use of if-elif-else construct."""

# ask user for an integer
n = int(raw_input("I'd like an integer please: "))

# analyze user's input (somewhat inaccurately)
if n > 0:
    print("You picked a positive number!")
elif n < 0:
    print("You picked a negative number!")
else:
    print("You picked zero!")
```

- Now let's look at conditions-2.py[10]:

```
"""Assesses the size of user's input. Demonstrates use of Boolean
 ANDing."""

# ask user for an integer
n = int(raw_input("Give me an integer between 1 and 10: "))

# judge user's input
if n >= 1 and n <= 3:
    print("You picked a small number.")
elif n >= 4 and n <= 6:
    print("You picked a medium number.")
elif n >= 7 and n <= 10:
    print("You picked a big number.")
else:
    print("You picked an invalid number.")
```

  # So here we have more branches, but now we have `#`, `>=`, and `and`, which all have equivalents in Scratch.

  # And we have to separate the conditions, since not all languages support expressions like `1 # n # 3`.

- What might positive.py[11] do?

[10] http://cdn.cs50.net/2015/mba/classes/5/src5/conditions-2.py
[11] http://cdn.cs50.net/2015/mba/classes/5/src5/positive.py

```
while True:
    n = int(raw_input("Please give me a positive int: "))
    if (n >= 1):
        break
print("Thanks for the positive int!")
```

# It looks like we're asking for integers from the user again, and only if the number we get is positive, do we do something called `break`. Since we're inside this `while True` loop that continues forever, we need to call `break` which will stop the loop. ( `break` also stops `for` loops, if we needed to.)

- It works as we'd expect, except, if we type in a name, we get a nice error:

```
% python positive.py
Please give me a positive int: Lin
Traceback (most recent call last):
  File "positive.py", line 2, in <module>
    n = int(raw_input("Please give me a positive int: "))
ValueError: invalid literal for int() with base 10: 'Lin'
%
```

# It looks like it's telling us the error is in line 2, and something about `int`, so we can deduce that the error is about assuming the input being an `int`, and not handling other cases.

- Another fun trick is to add this line:

```
#!/usr/bin/env python
```

to the top of files, so we can run them directly instead of `python hello.py`.

- We'll change `hello.py` to look like this:

```
#!/usr/bin/env python
print("hello, world")
```

- And then change the permissions of the files like this:

```
% chmod a+x hello.py
```

    # `chmod` is a program that changes the permissions of files, and `a+x` means that all, everyone, will have x, executable permissions, for the file `hello.py`.

- If we rename the file with a program called `mv` like so:

```
% mv hello.py hello
```

- Then we can just run it:

```
% ./hello
hello, world
%
```

- Next time we'll move to the web context, where we won't be printing to our screen but generating exciting HTML content.