LECTURE 2:
Wednesday August 30, 2023 9:00 PM

Loops.....

meow like a cat?

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

Why is this poorly designed? because it's not scalable at all. If we wanna add anymore, it gets messy.

Let's talk about while loops...

```
i=3
while(i != 0):
... print("meow")
... i = i-1
```

much more scalable! Just don't forget to update i - or infinite loop begins.

The code inside will execute while condition is true.

.... i = i - 1 important piece of code!

We're checking this condition at the start of every iteration because the code is executed.

Better convention... Start counting from zero!

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

effectively the same thing as i = i + 1. i += 1 and i -= does not exist

it is a better practice to count up, and to have the conditional expression be a "less than" one.

The "for" Loop and a list...

What's a List?
A list of things!

```
for i in [0,1,2]:
.... print("meow")
```

This is a list of integers. i is automatically initialized to 0, then 1... then 2...

Square brackets represent a list.

Problem with this solution though.
Again, it's not scalable. What if we go from three to a million?

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

returns a basic list [0, 1, ... x] where x is the value passed to range.

Pythonic improvement. If you don't use i, just use an underscore.

```
for _ in range(3):
.... print("meow")
```

moving forward...

```
print("meow"*3)
print("meow\n"*3)
print("meow\n"*3,end="")
```

→ "meowmeowmeow"
→ "meow" "meow" "meow" "
→ "meow" "meow" "meow"

Single line solution.
Not necessarily better, but a possible approach.

Loops are useful to get user input that fits certain criteria.

```
while True:
.... n = int(input("what is n? "))
..... if(n<0)
..... continue
..... else
..... break
```

We'll only escape this infinite loop until the user gives us a satisfying input.

```
while True:
.... n = int(input("what is n? "))
..... if n>0
..... break

for _ in rang(n):
.... print(n)
```

make a function

```
def meow(n):
.... for _ in range(n):
..... print("meow")
```

Simple function, prints n times.

```
def get_number():
.... while(true)
..... n = int(input("what's n?"))
..... if n>0
..... return n
```

We have the function once we get a good input.

```
def main():
.... number = get_number()
.... meow(number)
```

Hogwarts

```
students = ["Hermione", "Harry", "Ron"]
```

A list of strings!

How do we specify which student to print?

```
print(students[0])      "Hermione"
print(students[1])      "Harry"
print(students[2])      "Ron"
```

Lists are indexed starting at zero!

more efficient);

```
for student in students:
.... print(student)
```

Pay attention to the proper naming scheme!

We could also use numbers instead...

```
for i in range(len(students))
.... print(students[i])
```

We're accessing by indexing now

```
for i in range(len(students))
.... print(i+1, students[i])
```

explaining range(len(students)).
- range(x) expects x to be an int. it returns a list of [0, ..., x-1], x size.
- len returns the length of a list.
- we get a list [0, ..., n-1], where n is the size of students, allowing us to index through it.

Dictionaries! Dict

Data struct that allows you to associate a value with another value.
Hel-value pairs!

Let's keep track who is in what house?

```
students = ["Hermione", "Harry", "Ron", "Draco"]
houses = ["Gryffindor", "Gryffindor", "Gryffindor", "Syth"]
```

This does NOT work well using only lists
Dict to the rescue!

```
students = {"Harry": "Gryffindor"
            "Draco": "Slytherin"}
```

For matching! Better off this way.

Dicts allow you to look into via their keys

```
print(students["Hermione"])
print(students["Draco"])
```

This syntax returns the key of the dict!

more efficiently...

```
for students in students:
.... print(student, students[student])
```

key, value
or, student, house

What if we have more info about each of our students? e.g. the patronus.

	name	house	patronus
0	Hermione	Gryffindor	Otter
1	Harry	...	Stag
2	Ron	...	Dog
3	Draco	Slytherin	

implementation, key-values pairs

```
students = [
.... {"name": "Hermione", "house": "Gryffindor"
     "Patronus": "Otter"},
.... {"name": "Draco", "house": "Slytherin"
     "patronus": None}
]
```

Python keyword

3 keys, 3 values per entry in the list.
Now we have access

```
for student in students:
.... print(student["name"], student["house"]
           student["patronus"])
```

Nested Loops

recreate:

"mario bricks"

mario.py

```
for _ in range(3):
.... print("#")
```

Functions are useful for abstractions that help us understand what they do.

```
def print_column(height):
.... for _ in range(height):
..... print("#")

def print_row(width):
.... print("?" * width)
```

main() does not need to know underlying implementation of function.

How can we vary or block structure?

```
print_square(3)
```

```
def print_square(size):
.... for i in range(size):
..... for j in range(size):
..... print("#", end="")
..... print()
```

row of bricks
each brick in row
print brick
prints the newline, getting ready to start another row

Could also do

```
def print_square(size)
.... for i in range(size)
..... print("#" * width)
```

much more understandably by doing some basic abstraction.

These are the building blocks necessary to complete a vast majority of problems.