Introduction to Programming

Unit 3: Loops, lists and hashes

We'll start with "loops".

Loops are all about repeating tasks.

Repeating tasks without having to repeat code.

In other words, loops allow your programs to execute parts of its code multiple times.

Some loops allow your programs to execute parts of its code forever.

Other loops allow your programs to execute parts of its code for each item in a list of items.

And others loops allow your programs to execute parts of its code until a condition is met.

One is correct depending on your situation.

"Keep putting frosting on the cupcakes until there are no cupcakes left."

"Keep brushing your teeth until you've brushed all of them."

"Keep asking the user for a number until they guess the one you're thinking of."

"Keep asking the user for a word until they guess the Wordle of the day."

`while` loops

```
while condition:
    code
    code
```

"Keep putting frosting on the cupcakes until there are no cupcakes left."

"Keep putting frosting on the cupcakes while there are cupcakes that don't have frosting on them yet."

"While there are cupcakes that don't have frosting on them yet,
keep putting frosting on the cupcakes."

`while` loops look like `if` statements

```
while condition:
    code
    code


if condition:
    code
    code
```

`while` statements

```
code
code

while condition:
    code
    code

code
code
```

Notice the indentation in this code

```
code
code

while condition:
····code
····code

code
code
```

Conditions in `while` loops

```
user_input = int(input("Please enter a number between 1 and 10: "))

while user_input != number_to_guess:
    user_input = int(input("Please enter a number between 1 and 10: "))
```

Multiple conditions in `while` loops

```
user_input = int(input("Please enter a number between 1 and 10: "))

while user_input < 1 or user_input > 10:
    user_input = int(input("Please enter a number between 1 and 10: "))

print("Your number is " + str(user_input))
```

Multiple conditions in `while` loops

```
user_input = 0

while user_input < 1 or user_input > 10:
    user_input = int(input("Please enter a number between 1 and 10: "))

print("Your number is " + str(user_input))
```

Number guessing game with multiple guesses

Let's update our Number Guessing Game programs so that they let the user continue making attempts at guessing the number until they get it right.

Number guessing game with multiple guesses

```
while guess is incorrect:
    code
```

Wordle with multiple attempts

Let's update our Wordle programs so that they let the user make up to 6 attempts at guessing the word.

Wordle with multiple attempts

```
while guess is incorrect and has taken less than 6 guesses:
    code
```

Nested loops

A nested loop is simply a loop inside of another loop.

Nested loops

Nested loops are useful when you are repeating an action that is based on another repeated action.

Nested loops

For example, printing a grid with rows and columns.

Grid with rows and columns

```
while ...:
    while ...:
      print cell
```

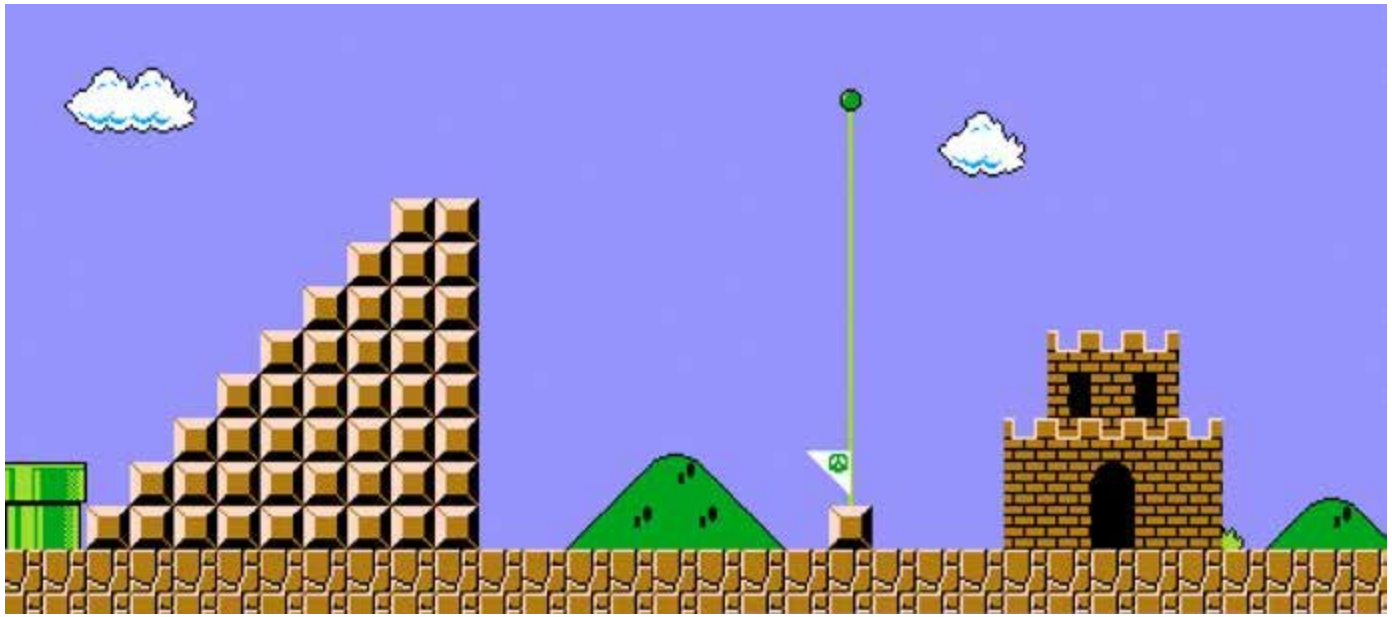
5 by 5 grid of x's

Let's write a program that prints the table below

```
X X X X X
X X X X X
X X X X X
X X X X X
X X X X X
```

- Print a table with 5 rows and 5 columns.
- Use two `while` loops.
- Hint: use `print("x", end="")` to print `"x"` without the new line.
- Hint: use `print("")` to print a new line.

Mario's wall jump



Mario's wall jump

`for` loops

`for` loops are another way to loop in Python.

`for` loops

It's easier to loop over a list using a `for` loop than a `while` loop.

`for` loops

```
for item in items:
    code
    code
```

Using a `for` loop with a string

```
first_name = "Marcos"

for letter in first_name:
    print(letter)
```

Using a `for` loop with a list

```
names = [
    "Abdul", "Ahmed", "Chichi", "Cindy"
    "Cristina", "Daniel", "Dimanche", "Hala"
    "Halima", "Ikran", "Isnino", "Joanna"
    "Margarita", "Marta", "Mercina", "Milad"
    "Mohammad", "Mugisha", "Mussie", "Mustafa"
    "Mustafa", "Silvana", "Tresor", "Zalmay"
]

for name in names:
    print(name)
```

Fizz buzz

- For the numbers from 1 to 100:
  - Print "Fizz" if the number is divisible by 3.
  - Print "Buzz" if the number is divisible by 5.
  - Print "FizzBuzz" if the number is divisible by both 3 and 5.
  - If the number is divisible by neither 3 nor 5, print the number itself.

Lists

Lists are values that contain multiple values.

Lists **store** other values.

Lists are a data type in Python.

We used a list in Wordle

```
from random import choice

word_options = ["slurp", "video", "right", "rusty", "rhyme",
                "enter", "minty", "nurse", "print", "sandy"]
#
#                                    ^
#                                    |
#                                    |
#           a list of strings -------+

word_to_guess = choice(word_options)
print(word_to_guess)

user_input = input("Please enter a word: ")
number_of_guesses = 0
```

Lists let you keep values that represent the same thing together.

Using a list can be better than separate variables.

Separate variables vs. lists

```
# A word per variable
word_option_1 = "slurp"
word_option_2 = "video"
word_option_3 = "right"
# ...

# vs.

# All words in a single variable
word_options = ["slurp", "video", "right", "rusty", "rhyme",
                "enter", "minty", "nurse", "print", "sandy"]
```

Lots of functions work with lists, for example `random.choice`.

Notice the square brackets and the commas

```
word_options = ["slurp", "video", "right", "rusty", "rhyme",
                "enter", "minty", "nurse", "print", "sandy"]
```

Lists can contain strings

```
friends = [ "Alec", "Bekah", "Ryan", "Sam", "Sean" ]
```

Lists can contain integers and floats

```
ages = [ 26, 59, 35, 23 ]
```

Lists can contain booleans

```
enabled = [ True, False, True, True, False ]
```

Lists can contain lists

```
tic_tac_toe_matrix = [
  [ "X", " ", "O" ],
  [ " ", "X", "O" ],
  [ " ", " ", "X" ]
]
```

Lists can contain a mix of data types

```
users = [
  [ "Marcos", 33, False ],
  [ "Ryan", 39, False ],
  [ "Bekah", 36, True ]
]
```

The three list operations you need to know about

- How to **access** an item in a list.
- How to **update** an item in a list.
- How to **add** an item to a list.

How to "access" an item in a list

To "access" an item in a list means to get one of the values stored in the list.

Accessing an item in a list

We use `[ ]` (square brackets) to access an item in a list, and we pass the index (*the position*) of the item we want to get.

The index

"Index" means the position of the item in the list (eg, first, second, third, etc.)

Accessing an item in a list

```
word_options = ["slurp", "video", "right", "rusty", "rhyme",
                "enter", "minty", "nurse", "print", "sandy"]

print(word_options[0]) # prints slurp
print(word_options[1]) # prints video
print(word_options[5]) # prints enter
```

Item index offsets

Notice that the first item in a list is in index 0.

How to "update" an item in a list

Updating a item in a list means we're changing the value contained at a specific index.

Update an item in a list

```
word_options = ["slurp", "video", "right", "rusty", "rhyme",
                "enter", "minty", "nurse", "print", "sandy"]

print(word_options[5]) # prints enter

word_options[5] = "pizza"

print(word_options[5]) # prints pizza
```

How to "add" an item to a list

Adding an item to a list means we're making the list bigger by adding a value to it.

Adding an item to a list

```
word_options = []

word_options.append("slurp")
word_options.append("video")
word_options.append("right")

print(word_options[1]) # prints video
```

Adding an item to a list

Values are appended to the end of the list.

Adding an item to a list

You can add an item to the beginning of a list, but we'll just stick with `append` for now.

Looping over a list

```
friends = [ "Alec", "Bekah", "Ryan", "Sam", "Sean" ]

for friend in friends:
    print(friend)
```

`break` and `continue` are statements that give us additional control over how loops work.

`break` is used to stop the loop.

A `while` loop will stop when its condition is `False`,
`break` lets us do the same.

`continue` is used skip to the next iteration.

`break` and `continue` example

```
while True:
    if the player has won the game:
        print("You win!")
        break
```

`break` and `continue` example

```
while True:
    if this is an item we want to skip:
        continue
```