Notes by Regina Gerbeaux

CS106A Notes - Python
Chapters 1-5 of Karel Reader

INTRODUCTION TO BASICS OF PYTHON

• To create a **comment**: use the `#` sign.

```
#this is a comment lalalalala
```

• To **import** a package to access functions, standard operations, etc.:
`from location-of-package-goes-here import *`

```
from karel.stanford import *
```

This imports the `karel.stanford` library definitions and operations.

• To define a new **function**:
  • Start with defining the function using `def`. Specify the name of your function (e.g.: `main`).
  • Add a pair of parentheses and colon following the name of your function.
  • Add the body of your function definition indented in so Python knows that code is part of the function. (Called **code block**.)
  • The body of your function should use imported commands.

```
def main():
  body of function
```

• To define a new **command**:
  • Start with defining the command using `def`. Specify the name of your command (e.g.: `name`).
  • Add a pair of parentheses and colon following the name of your command.
  • Add the body of your command definition indented.
  • The body of your function should use commands that you've imported.

```
def name():
  body of function
```

- A few notes:
  - You should define functions written one after the other.
  - You can't define a function inside another function, unlike JS.
- Why make multiple functions?
  - Functions are **reusable blocks of code**. If you write out your code and see lots of repetitive parts, you might be able to write one function to cover those repetitive parts, so instead of writing the same code 5 times, you simply write the code 1 time and run *that* function in place.
  - For example, if our main function looks like this:

```
def main():
  turn_left()
  turn_left()
  turn_left()
  move()
  turn_left()
  turn_left()
  turn_left()
  move()
```

Do you notice how we have a lot of repeating code? We could probably condense this!

  - We can simplify it as such:

```
#define main function
def main():
  turn_right()
  turn_right()

#turn right and move
def turn_right():
  turn_left()
  turn_left()
  turn_left()
  move()
```

Notice in `main()`, we call `turn_right()` twice.

By writing this function, we limit the overall amount of code we write, therefore lowering the potential for typos/errors and saving us keystrokes.

BASIC FOR LOOPS

- What if we want to do a task many, many times? For example, if we want a code to run 40 times, should we type that line of code…40 times?
- As you may have guessed—nope. There's a shortcut for that! It's called a **loop**.
- `for` loops allow us to write *one* code block and create a **condition** specifying how many times that code block should run.
- To define a `for` loop:
  - Start your code with `for`.
  - Create a counter for the loop (for example, `i`.)
  - Follow the text with the phrase `in range`, followed by parentheses that take in a number parameter. Finish the phrase with a colon.

```
for i in range(count):
  code block to repeat
```

- So for example:

```
def main():
  move()
  for i in range(15):
    put_beeper()
  move()
```

This will run `move()`, then run the `for` loop 15 times (which means `put_beeper()` will run 15 times), and then run `move()` one last time.

POSTCONDITIONS WITH PRECONDITIONS

- You can add more code within your `for` loop! In the example above, we have only one line of code, but do know your loops can have as many lines of code as you want.
- When you have many lines of code *within* a loop, however, you have to make sure that once your loop executes once, it is set up to have a **valid** state to restart the loop. This means that you need to make sure when the loop starts again, it will be able to execute the loop once again!

- For example, let's say you want to create a function to withdraw some money from an ATM.
  - We want to withdraw $30 in bills of $10 each.
  - In order for the loop to work, there has to be more than $30 in the ATM as a **precondition**.
  - Otherwise, if there is only $10 in the ATM for example, we'll get an error when the `for` loop tries to execute the second time.
  - This is because there's no more cash left, meaning the **postcondition** of the loop is that there has to be another $10 available to dispense.

```
def main():
  goToAtm()
  for i in range(3):
    withdraw(10)
  leaveAtm()
```

Here, we go to the ATM. Our `for` loop tells us we will withdraw $10 each time it runs (for a total of `3` times.)

NESTED LOOPS

- You can also have loops exist in other loops.

This `for` loop will run as the standard operation for what happens when you see an attendee at the party. There are 20 people at the party.

```
# executes flow at a party
def main():
  for i in attendee(20):
    greet() #says hello to attendee
    pleasantries()
    eat()
  exit() #says goodbye and leaves party

# executes the contents of pleasantries
def pleasantries():
  howAreYou()
  whatsNew()
  youLookGreat()

# executes steps to eating at party
def eat()
  for i in hunger(2):
    goToKitchen()
    getFood()
    goBackOutside()
```

`pleasantries()` calls the function below

These functions will execute when `pleasantries()` runs! (This example assumes these functions already exist.)

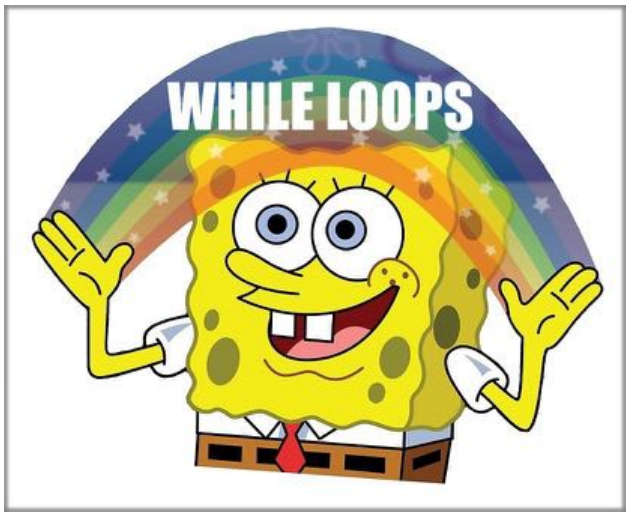
`eat()` is a loop inside a loop! It will run these functions twice after you exchange `pleasantries()`, since you stress-eat when you're overwhelmed.

CS106A Notes - Python
Chapters 6-9 of Karel Reader

Chapter 6: While Loops

INTRODUCTION TO `WHILE` LOOPS

- We learned in the previous chapter about `for` loops. However, `for` loops are great when we want something to repeat for a *predefined* number of times.
  - This means `for` loops are great if we know how many times we want an action to happen (say, if we wanted Karel to place down 15 beepers!)
  - But what if we *don't* know how many times we want a piece of code to run?
- That's where `while` loops come into play!
- `while` loops run **while a certain condition is true**.
- For example:




```
#my dog wants lots of pats
def main():
  #call the pat function
  pat_pat_pat()

#we have to define our pat function!
def pat_pat_pat():
  #repeat while this condition holds
  while dog_is_here():
    pet_the_dog()
```

- In the above example, we want to give our dog lots of pats. I can only pat my dog while she's here, though.
- So I set a **condition** where as long as my dog is here (`while dog_is_here()`), I can pet her (`pet_the_dog()`).
- So that way, if my dog is here for 5 minutes, or 15 minutes, or 3 seconds, I'll *continue petting her as long as she's here*.

Phew. Good thing she's napping away from me right now. Or else I couldn't make these notes.

- This makes our code more flexible, because we're not setting a *predefined* number of times to execute the code. Again, we repeat the code **as long as the condition is true**. (Imagine what a travesty it would be if I only pet her 5 times, and she was here for longer! Also, imagine if I set my mind to pet her 100 times, but she ended up seeing a squirrel and running away—I would be foolish to continue a petting motion without her being here!)

FORM OF A WHILE LOOP

- `while` loops follow this logic:

```
while this condition is true:
    repeat these functions!
```

- When a `while` loop runs, the program will check to see if your condition (also known as a **test**) is true or not. If it's true, the code will run! If it's not, the loop will end and the program will move onto the next step in its `main` function (if there's anything else.)

FENCEPOST BUG

- Sometimes, when we define a `while` loop, the test no longer passes but we still want something to be executed.
- *The Karel example in the chapter is a fantastic one (where Karel needs to place 7 beepers, but can only move 6 times.)* I highly recommend you try that example out.
- When we want something to execute one last time after the test no longer evaluates as true, we call this a **fencepost error**.
- To solve the issue from Karel, all you have to do is add the action you want done one last time *outside* of the loop!

```
def main():

    # repeat until Karel's path is no longer clear
    while front_is_clear():
        # place beeper on current square, then move
        put_beeper()
        move()
    # solve the fencepost bug!
    put_beeper()
```

This is the `while` loop!

Since we have to place 1 more beeper, we add a `put_beeper` function outside of the `while` loop to solve the fencepost error.

Chapter 7: If Statements

INTRODUCTION TO CONDITIONAL STATEMENTS (`IF` AND `IF/ELSE`)

- `if` statements are pieces of logic that will run code *if* a certain condition is met. `if` that condition is not met, then the code will not execute at all.
- For example, let's say my dog wants some treats. ("Treats?!") I will only give her treats `if` she has been a good gurl.

```
#start of program
def main():
  if leia_is_good():
    feed_treat()
```

Leia gets a treat if she is good. :)

- What about `if/else` statements? `if` a condition is met, that code will run. If it is *not*, the code within the `else` block is run.
- So another example: my dog loves chasing squirrels, and she likes sniffing everything. `if` a squirrel is present, she will chase the squirrel. Or `else`, she will continue sniffing everything.

```
#start of program
def main():
  while outside():
    trot()
    do_outside_things()
  #prevent fencepost bug
  trot()

#what outside things will Leia do?
def do_outside_things():
  #if there are squirrels, she becomes a hunter!
  if squirrel_present():
    chase_squirrel()
  else:
    sniff_everything()
```

As long as Leia is outside, she will trot + do outside things. :)

The last thing Leia should do is trot - she still has to go home!

These are the outside things that Leia does!

LIST OF CONDITIONS FOR KAREL

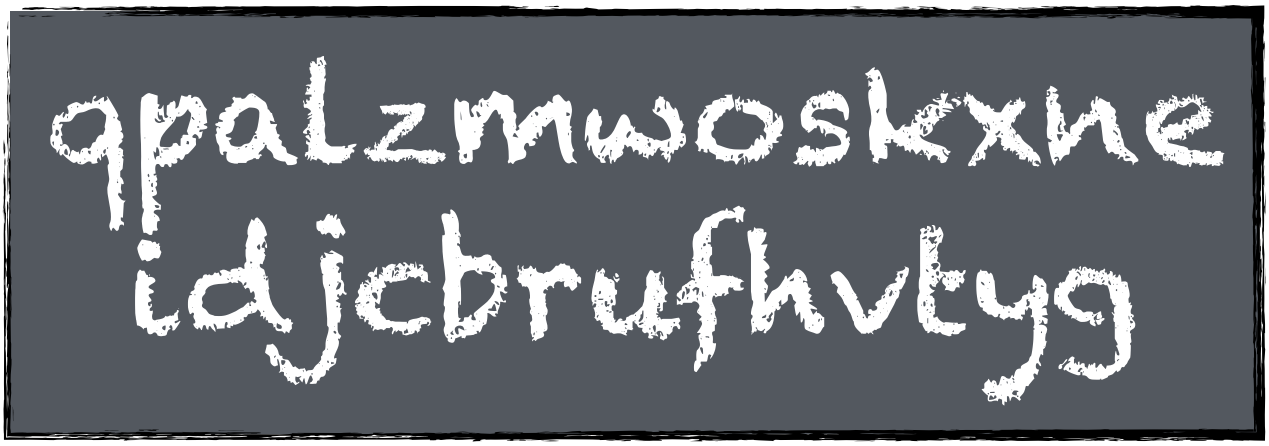
• Here are the conditions that Karel knows of! (Table from Karel Reader handout):

Test	Opposite	What it checks
`front_is_clear()`	`front_is_blocked()`	Is there a wall in front of Karel?
`beepers_present()`	`no_beepers_present()`	Are there beepers in this corner?
`left_is_clear()`	`left_is_blocked()`	Is there a wall to Karel's left?
`right_is_clear()`	`right_is_blocked()`	Is there a wall to Karel's right?
`beepers_in_bag()`	`no_beepers_in_bag()`	Does Karel have any beepers in its bag?
`facing_north()`	`not_facing_north()`	Is Karel facing north?
`facing_south()`	`not_facing_south()`	Is Karel facing south?
`facing_east()`	`not_facing_east()`	Is Karel facing east?
`facing_west()`	`not_facing_west()`	Is Karel facing west?

Chapter 8: Refinement (putting to use what you've learned so far with our friend Karel!)

REFINE YO' CODE (TOP-DOWN METHODOLOGIES)

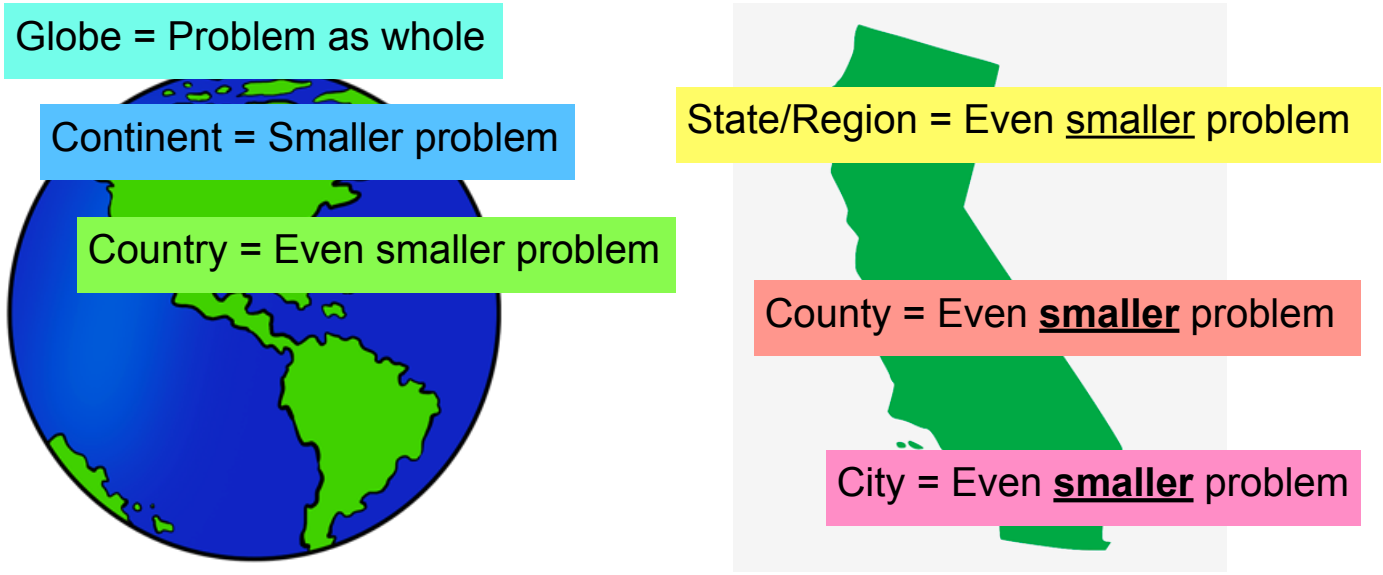
- The beauty of code comes from being able to take a large problem and breaking it down into sizable chunks that are easier for you to solve.
- The concept of **software engineering** comes from **a set of programming methodologies**. This means there are "good practices" so that other engineers can read and understand your code.



Imagine if you had to learn the alphabet with no order, every time! That would make the alphabet really hard to read and learn.

*Ah! Much better! Having **good practices** is part of being a good software engineer, so everyone understands you more easily.*

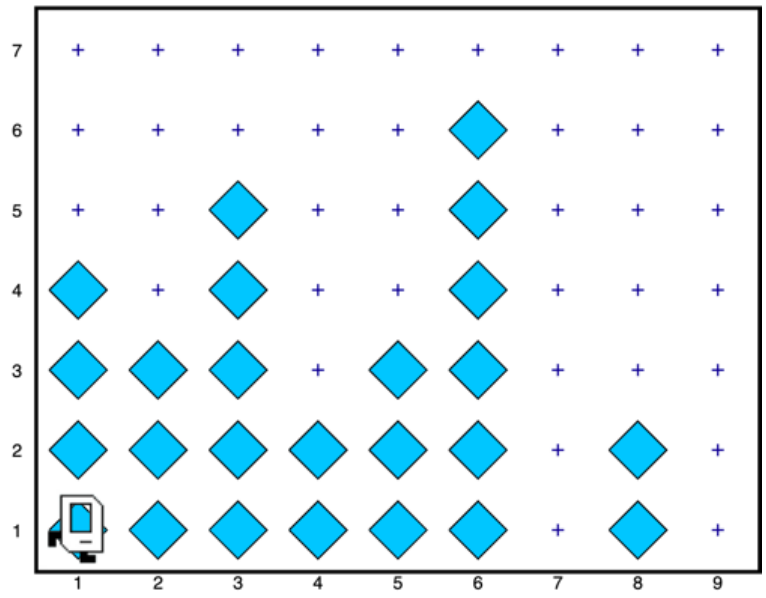
- One of the best strategies to adopt right now is **top-down design** (also known as **stepwise refinement**.)
  - Start with the problem, then break it down further if necessary



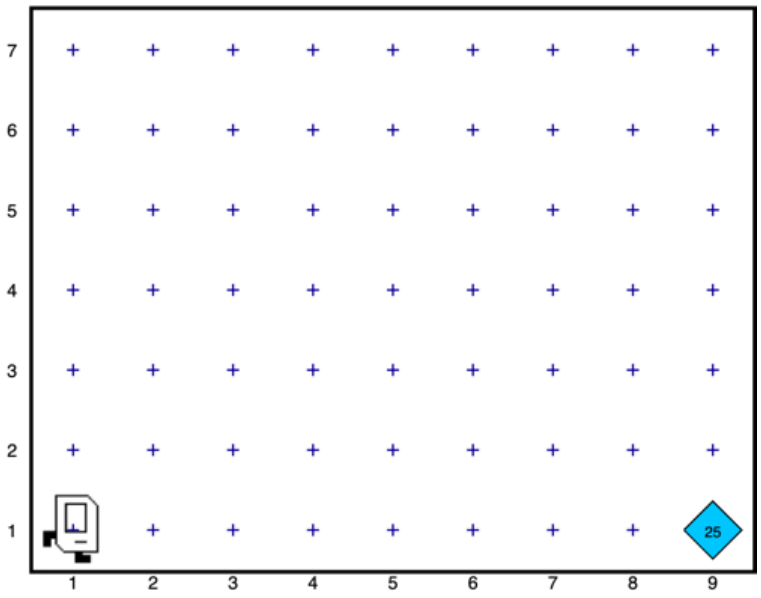
For example, let's treat an assignment, or a problem, like it's the whole world. That seems like a huge endeavor — so try breaking it down as much as you need to. You can go as small as a state, county — even a neighborhood within a city, or even a street!

TOP-DOWN IN CODE

- **How do we do this with code?**
  - It helps to look at your **starting point** and then figure out what your **end point** is. In the Karel Reader, this is your assignment:



Start



Finish

- We see that the tasks are to collect the beepers, drop them in the bottom right corner, and return home. (**Note: when we break things down into human-level instructions, this is called *pseudocode* and can serve as a fantastic blueprint for what has to be done.**)
- So, we can write this as your main code:

```
# highest level of what your code is accomplishing

def main():
  collect_all_beepers()
  drop_all_beepers()
  return_home()
```

TO ITERATE IS TO IMPROVE (ITERATIVE TESTING)

- While you're building your code, it is imperative to **test as you go!** It makes debugging (looking for errors) *much* simpler than if you built the whole thing and your program crashes (then you have no idea where it went wrong, and you might have to change *a lot* of your code to accommodate a mistake that could have been caught earlier on.)
- From our example above, let's say we want to **test** a single tower for Karel to pick up all the beepers. Recall our first step is `collect_all_beepers()`. That is a large task — so let's *size it down* (top-down approach!) We can start with Karel collecting just one tower of beepers.
- Here is our step, broken down:

```
# break down collect_all_beepers() into just
collecting one tower of beepers!

def collect_all_beepers():
  # here's our test to make sure it works
  collect_one_tower()
  move()
```

- Great! Now we have tested to see if we can collect just one tower. If that works, we can move on. But how do we get Karel to collect one tower? (Here, we *size that down* even more, following the top-down approach.)
- Let's pseudocode it:
  - Karel should face north to collect a tower.
  - Karel picks up its line of beepers.
  - Karel is done picking up the beepers! Time to turn around.
  - Karel goes back to the wall where it started. Remember, Karel is facing south now. We'll have to prep it to move to the next tower.
  - Karel should turn to face the next tower.
- Phew! If we can pseudocode it, we're ready to make it into code. Let's go!

```
# define how to collect_one_tower()!

def collect_one_tower():
  # Karel has to turn left so it's facing north
  turn_left()
  # Karel can collect its beepers now!
  collect_line_of_beepers()
  # All done – turn around Karel
  turn_around()
  # Go back to the wall!
  move_to_wall()
  # reorient Karel to face the next tower!
  turn_left()
```

- Woohoo! Karel can (hopefully) now pick up one tower of beepers. Be sure to test it before moving on!

FUNCTION PRECONDITIONS AND POSTCONDITIONS

- Remember preconditions and postconditions from Chapter 5? Preconditions are the conditions that have to be true for a function to start. Postconditions are the conditions that have to be true after a function finishes.

- In our problem above, notice how Karel has to face the right direction in order to proceed with collecting the tower. That's our precondition.

```
# here is our precondition!

def collect_one_tower():
  # Karel has to turn left so it's facing north
  turn_left()
```

- Similarly, Karel has to reorient itself to finish collecting the current tower of beepers and to prepare itself to collect the next tower of beepers. That is our postcondition.

```
# here is our postcondition!

def collect_one_tower():
  # reorient Karel to face the next tower!
  turn_left()
```

REPEATING THE PROCESS

- Yay, you wrote that code that will make Karel collect one tower! You're awesome!
- Now, we have to help Karel collect all the beepers. We have to…do it again.

I'm a big fan of Foster's Home for Imaginary Friends from Cartoon Network. This is Cheese. Look it up on Youtube.

• Recall that our code from the `main()` function looks like this:

```
# highest level of what your code is accomplishing

def main():
  collect_all_beepers()
  drop_all_beepers()
  return_home()
```

• Now that we wrote `collect_one_tower()`'s logic, the next step would probably be to write a `while` loop in order to help Karel collect towers while Karel is not facing a wall.
• Thus, our logic would look something like this:

```
# while loop in collect_all_beepers()

def collect_all_beepers():
   #as long as Karel is not facing a wall, it
   should collect the beepers in the tower.
  while front_is_clear():
    collect_one_tower()
    move()
   # don't forget about the fencepost bug!
  collect_one_tower()
```

FINAL PRODUCT

• So now that we've discussed how to break down code and implement loops that serve what we're trying to help Karel do, here is the final solution to the problem from above:

```
from karel.stanfordkarel import *

def main():
   collect_all_beepers()
   drop_all_beepers()
   return_home()

def collect_all_beepers():
   while front_is_clear():
      collect_one_tower()
      move()
   collect_one_tower()

def collect_one_tower():
   turn_left()
   collect_line_of_beepers()
   turn_around()
   move_to_wall()
   turn_left()

def collect_line_of_beepers():
   while beepers_present():
      pick_beeper()
      if front_is_clear():
         move()

def drop_all_beepers() :
   while beepers_in_bag():
      put_beeper()

def return_home():
   turn_around()
   move_to_wall()
   turn_around()

def move_to_wall():
   while front_is_clear():
      move()

# Turns Karel 180 degrees around
def turn_around():
   turn_left()
   turn_left()
```

Karel collects all beepers, drops them off, and then returns home.

Collects beepers from every tower.
Precondition: front has to be clear.
Postcondition: Karel must be facing east in the easternmost corner of the first row.

Collects beepers from one tower and return to starting position.
Precondition: Karel must start on 1st row facing east.
Postcondition: Karel must be facing east in that same corner.

Collects beepers.
Precondition: There is a beeper in front of Karel for it to pick up.
Postcondition: No more beepers are present in a corner.

Drops off beepers.
Precondition: There must be beepers.
Postcondition: Karel must have no beepers left in its bag.

Karel returns home.
Precondition: Karel must face east somewhere on the first row.

This moves Karel forward until it is blocked by a wall.

This helps Karel turn around to face the opposite direction.

Chapter 9: Extra Karel Features

WHAT ELSE CAN KAREL DO?

- Karel can also paint in different colors:
  - `BLANK` (removes color on the square)
  - `BLACK`
  - `BLUE`
  - `CYAN`
  - `DARK_GRAY`
  - `GRAY`
  - `GREEN`
  - `LIGHT_GRAY`
  - `MAGENTA`
  - `ORANGE`
  - `PINK`
  - `RED`
  - `WHITE`
  - `YELLOW`
- To paint in different colors, write the following code:
  - `paint_corner(`*`color name goes here`*`)` - Karel will paint the color it is standing on the color name you've entered.

```
#paint the corner blue!
def make_it_blue( ):
  paint_corner(BLUE)
```

This will paint the corner blue.

- To check the color of the corner, write the following code:
  - `corner_color_is(`*`color name goes here`*`)` - Karel check the the corner color is the color name you've entered.

```
#is the color of the corner blue?
def color_check( ):
  corner_color_is(BLUE)
```

This will be either true or false.

CS106A Notes - Python
Week 1 Notes: Introduction to Karel (4/13/2020 Lecture)

INTRODUCTION TO KAREL
- Karel is a robot! Karel speaks Python. :)
- Karel the Robot lives in a world called Karel's World. It's a pretty simple world — it's got walls that Karel can't walk through, and beepers that Karel can pick up and put down.
- Karel knows **four** commands:
  - `move()` - moves Karel over *1 unit* in the direction Karel is facing.
  - `turn_left()` - Karel turns 90 degrees to the left (counter-clockwise)
  - `pick_beeper()` - Karel can pick up *1 beeper*
  - `put_beeper()` - Karel can put down *1 beeper*
- Our view of Karel is from the Bird's Eye view: It means you're looking at Karel from the top-down.

INTRODUCTION TO PYCHARM
- We used an IDE for Assignment 0 to move Karel through Karel's World and to put down and pick up beepers.
- For the rest of class, we'll be working with **Pycharm**, which is also an IDE (stands for *integrated development environment*)
- We can run Karel's World (and any code we write for Karel to do) in Pycharm.
- An IDE has three parts: It has:
  - A *text editor* where we can write and edit our code.
  - A *terminal* (bottom left on PyCharm) - to run anything in python, you prepend your statement with `python`. (So if you wanted to run the file `MyApp.py`, you would type `python MyApp.py`.
  - *Boilerplate code* at the bottom that launches your code when you type "`python name-here.py`" in the terminal. It looks like this:

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

WHAT ARE FUNCTIONS?

- **Functions** are statements that add new commands to Karel's vocabulary.
- Recall from above that Karel knows the four commands (or functions): Karel knows how to move, how to turn left, how to pick up beepers, and how to put down beepers.
- What if we want to make Karel do something else? Well, we can define a new function to make Karel do so!
- Here is the typical format for defining new functions:

```
def name-goes-here():
  what happens goes here
```

- So let's say we want to make Karel turn right. We would do so like this:

```
def turn_right():
  turn_left()
  turn_left()
  turn_left()
```

Notice we define new functions with the functions Karel already knows how to do. Recall that Karel knows how to turn left, so we tell Karel turning right is the same thing as `turn_left` three times.

- Great! Now we have five commands that Karel can work with: `move()`, `turn_left()`, `pick_beeper()`, `put_beeper()`, and our newest addition: `turn_right()`.

GOOD CODING STYLE (COMMENTS)

- As you write your code in the text editor, it is helpful to write **comments** as you go along. (To see more about comments and how to write them, reference Week 1A's notes in another document.)

ANATOMY OF A PROGRAM

- So what makes up a **program**?
  - First, you have your **import packages**.
    - This is the stuff you import in to make your code work. For example, Python doesn't know what *Karel* is, but if you type the

statement `from karel.stanfordkarel import*`, this will bring in that file and make it available for use in your current Python file.

- Then, we have our **main functions** (also known as **source code**.)
  - Executes the main functions in our code, one line at a time, from top to bottom.
  - The body of our function are made up of commands, and those commands should be indented by 1 (so that Python understands they are part of the main function.) - This is called your **code block***.
    *Note: *all code blocks are detonated by a colon* (`:`).
- Next, we have our **helper functions**.
  - We define new vocabulary here. (So for example, the `turn_right()` function would belong here as a helper.
  - You can define as many helper functions as you'd like.
- Finally, we can **start** our **program.** (The boilerplate code we have at the bottom.)

- Here's a full example of the anatomy of a program!

```
from karel.stanfordkarel import*

def main():
  move()
  pick_beeper()
  move()
  turn_left()
  move()
  turn_right()
  move()
  put_beeper()
  move()

def turn_right():
  turn_left()
  turn_left()
  turn_left()

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

We `import` Karel here!

This is our **main function**. Notice the indentation to the commands under `main` - this tells Python that these are the commands inside this function.

This is a **helper function**. It gives us new vocabulary to use in our `main` function.

Finally, we start!

CS106A Notes - Python

Week 1 Notes: Control Structures in Karel (4/15/2020 Lecture)

INTRO TO CONTROL FLOW

- Karel's World is formed as a **grid**.
  - **Streets** run East to West.
  - **Avenues** run North to South.
  - **Corners** are made up of a street and an avenue intersection.
    - So if Karel is standing in the southwest-most grid, it would be standing on 1st Street, 1st Avenue.
    - If Karel moves forward, because Karel is facing east, Karel would then stand on 2nd Street, 1st Avenue.

- Karel's beeper bag can have 0, 1, or more (up to infinite) beepers. **Your assignments will always tell you how many beepers Karel is starting out with.**

PROGRAMMING STYLE: COMMENTS

- Recall that **comments are words that you want humans to read, but that you want the computer to ignore.**
  - For example, we use **comments** to explain what our code is doing.
- Previously, you learned that hashtags (#) create a comment. These are **one-line comments**.

```
#lalala this is a one-line comment
```

- If you need to write a **multi-line comment**, use three double quotation marks:

```
"""
File: ExampleKarel.py
---------------------
This is a multi-line comment. We use the
double quotation marks to write multiple
lines of comments.
Recall from your Karel Reader that it is
good practice to make your code easy for
other engineers to read and understand!
Comments help shine light on what our code
in the file is attempting to accomplish.
"""
```

PROGRAMMING STYLE: NAMING CONVENTIONS

- To name your functions, pick **descriptive names**.

```
def eat():
  while there_is_still_taco():
    grasp_one_taco()
    lift_taco_to_mouth()
    open_mouth()
    put_taco_in_mouth()
    take_bite_in_taco()
    put_remaining_taco_down()
    eat_taco()
```

This code is not fully complete (I would have extra functions to specify how much of the taco to bite into, how much chewing to do in `eat_taco`, etc.), but **you understand what the functions are trying to accomplish based just on their names.**

- Additionally, notice the way we write names of functions that **have more than one word in the name**.
- We add **underscores** between the words, so that it is easy to read the separated words. All words are **lowercased**. This is called **snake_case**.
- There are several ways within computer programming to name our functions. We use snake casing in Python.


```
def there_is_still_taco
```

Correct!

```
def thereisstilltaco
```

Incorrect.

- Note: If you have programmed before, you might recall something called camelCasing (for example, in JavaScript.) *We don't use camelCasing for our functions in Python.*

Software Engineering Principle: Aim to make programs readable by humans!

That's why we use these conventions — if everyone is on the same page with how to write code, we can understand each others' code much more easily!

`for` Loops

- A `for` loop's **syntax** looks like this:

How many times you want the code to repeat

```
for i in range(number goes here):
    repeated code goes here (body)
```

Code that is **indented** is part of the `for` loop.

Don't forget the colon!

- For example, let's say you're sleep deprived, and you want to drink five cups of coffee (you wild animal!)
- You'll want to define your function, `fight_sleep_deprivation()`.
- Then, you'll want to specify a `for` loop where you `make` and `drink_coffee` five times (did I mention you're crazy?!)
- So, you would specify your range as `5` and the `coffee` loop would happen…you guessed it! 5 times!
- Let's see this in action:

Indentation!

```
def fight_sleep_deprivation():
    for i in range(5):
        make_coffee()
        pour_coffee()
        drink_coffee()
```

This happens five times!

PRECONDITIONS AND POSTCONDITIONS

- You have to make sure the code you're starting with (**precondition**) is set up to let the `for` loop run.

  - In our coffee example, our precondition is that **there must be coffee ready to be made.** (That's why we start our loop with `make_coffee()`!)
- You also have to make sure the code you're ending with (**postcondition**) is set up to work again when the `for` loop runs again.
  - In our coffee example, our postcondition is that your coffee cup should be empty, so you can make more coffee. If you haven't drunk your coffee and you make and pour more, your coffee cup will overflow!

`while` Loops

- Remember, `for` loops are great when we know *how many times* we want something to happen. That's why we have to predefine the number of times the loop will run within the `range`.
- What if we want something to run *as long as a certain condition is met*? That's where `while` loops come in! `while` loops are great for when you don't *know* how many times the loop should run, but you want the loop to run as long as a condition is true.
- The `while` loop **syntax** looks like this:

```
while condition goes here:
  repeated code goes here
```

As long as this statement is **true**, the code will run. The code stops running when it's **false.**

- Condition is checked at the start of the code, and each time it's about to repeat.

HOLD MY COFFEE, KAREL

- So, let's continue with our coffee example. (I love coffee.) Let's say ***you're really wild*** and you're not going to predefine the number of cups of coffee you're going to have. You'll just drink coffee until you're no longer sleepy! (Or until you die, I guess.)

- That's when you would use a `while` loop!
- Your code would look something like this:

```
def fight_sleep_deprivation():
  while still_sleepy():
    make_coffee()
    pour_coffee()
    drink_coffee()
  go_on_with_day()
```

This code gets looped.

As long as you're still sleepy, you'll make, pour, and drink coffee. You stop when you're no longer sleepy (or dead, but you can't be sleepy if you're dead.)

Notice this isn't part of the while loop! This executes after we're done with the loop.

Right now would be a fantastic time to plug one of my favorite coffee songs: Le Cafe - Oldelaf on YouTube: https://www.youtube.com/watch?v=UGtKGX8B9hU

- You would continue to fight sleep deprivation as long as the condition "still sleepy" is still true. As soon as it evaluates as *false* (you are no longer sleepy!), you can go on with your day.
- Be aware of **fencepost bugs** (that you can read about in my notes on Chapter 6 of the Karel Reader!) *Also known as "Off by One Error"* since the error typically occurs when a conditional is no longer met, but we want something to happen *one last time*.

`if` Statement

- We use `if` to **check whether a condition is true or not**. If it is true, it executes. If they are not, it will not execute.
- The difference between an `if` statement and `for` / `while` loops:
  - **An `if` statement will only execute when the condition is true. It moves on if it is not true.**
  - **`for` and `while` are loops: they will repeat until the condition is false.**
- Here is the **syntax** for an `if` statement:

```
if condition goes here:
  code here will execute
```

If this condition is true, then the code will execute.

Code will not execute if condition evaluates to false.

- For example, let's say you're a CS106A student and you have an assignment coming up that's due soon. *But…* you don't exactly have great study habits and you always wait until the night before to finish everything. (Sorry, Prof. Piech & Prof. Sahami.)
- Not to worry: we can express this in an `if` statement!
- Your pseudocode would go something like this:
- `if` the assignment is due tomorrow, do the assignment.


```
def homework():
  if due_date_is_tomorrow():
    do_karel_assignment()
```

Our function `homework()` will run if the due date is tomorrow. (You procrastinator!) :)

`if-else` Conditionals

- `if` statements are great for checking if something should run or not. But what if there's *something else* we want to happen if the conditional is *false?*
- That's where `if-else` statements come in! `if-else` **statements read that <u>if</u> the condition is true, the code within the `if` block will execute. Otherwise, the code within the `else` block will execute!**
  - In other words, if the conditional is <u>true</u>, the code under `if` runs.
  - If the conditional is <u>false</u>, the code under `else` runs.
- Let's go back to our example from above!
- You know you want to wait until the last minute to do your homework (hey, no judgement — but maybe you should start sooner? Never mind.)
- *But what will you do during the time you aren't doing your homework?*
- That's where the `else` statement comes in handy. Since the due date isn't tomorrow, **THINK OF ALL THE THINGS YOU COULD DO INSTEAD.**

- So, `if` the assignment is due tomorrow, do the assignment and feel complete regret for procrastinating. `else`, play all the video games in the world, make cool memes, eat junk food, and pet your dog.

```
def homework():
  if due_date_is_tomorrow():
    do_karel_assignment()
    feel_regret_for_procrastinating()
  else:
    play_video_games()
    make_cool_memes()
    eat_junk_food()
    pet_dog()
```

Notes by Regina Gerbeaux

CS106A Notes - Python
Week 1 Notes: Decomposition (4/17/2020 Lecture)

Recap on 4/15/2020 Lecture

- **Functions** are pieces of code we can write to make things happen (like commands!)
  - For example, Karel knows his four basic commands: `move()` `turn_left()` `put_beeper()` `pick_beeper()`
  - From those commands, you can create new commands!
  - Here's an example from lecture:

```
def main():
    go_to_moon()

def go_to_moon():
    build_spaceship()
    # a few more steps

def build_spaceship():
    # how to do this?
    put_beeper()
```

Karel doesn't know the commands `go_to_moon()` or `build_spaceship()`, but it does know `put_beeper()`! Now, we've taught Karel that to build a spaceship, all it has to do is put down a beeper.

Any fans of The Regular Show from Cartoon Network here? If not, I highly recommend. Season 1 Episode 1 is on YouTube for free. You're welcome.

- Use **`for` loops** to run a piece of code a ***predefined*** *number of times*. So, these are great for when you know how many times you want your loop to run!
  - For example:

```
def main():
  put_beeper()
  put_beeper()
  put_beeper()
  put_beeper()
  put_beeper()
  put_beeper()
  put_beeper()
  put_beeper()
  put_beeper()
```

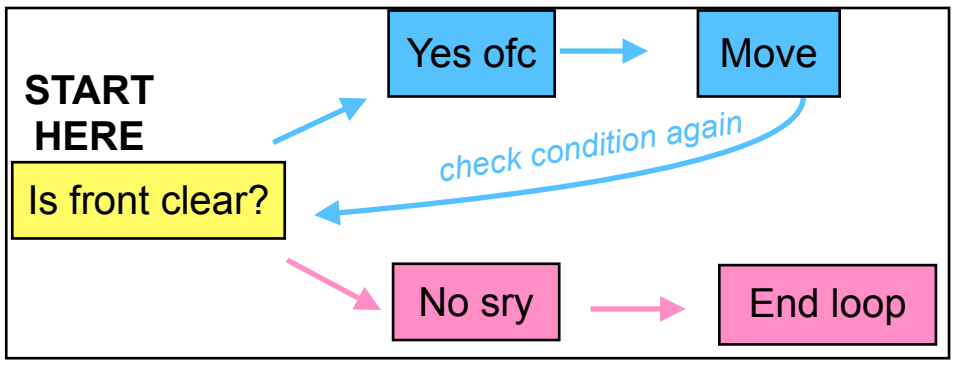
Ew?

```
def main():
  #much better!
  for i in range(9):
    put_beeper()
```

YUS.

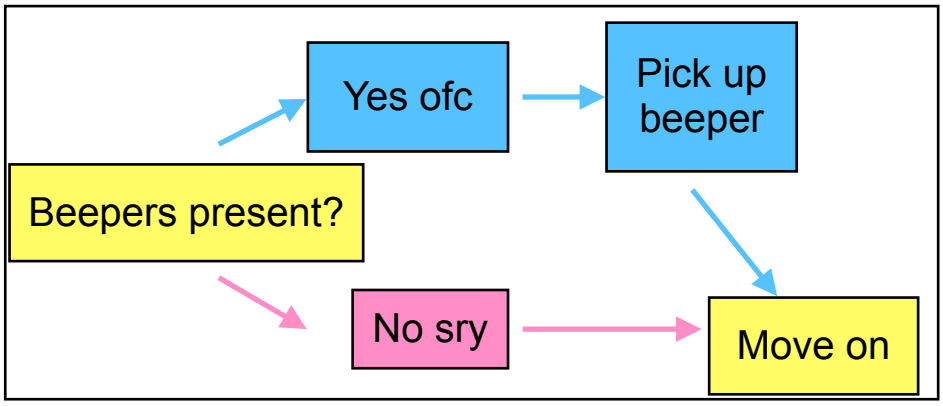
  - *Imagine if we had to put down 99 beepers! See why the `for` loop is useful in this case?*
- Use **`while` loops** to run a piece of code *as long as a **condition** is true.* These are great for when you don't know how many times you want your loop to run.
  - `while` loops will evaluate whether the condition is true at the beginning of the loop, and at the end of the loop.
  - The condition must be true for it to run. If it's not true even from the get-go, the code will *not* execute!
  - For example: (yellow is condition checking, blue is the loop, and pink is the exit of loop)

```
def main():
  while
front_is_clear():
    move()
```



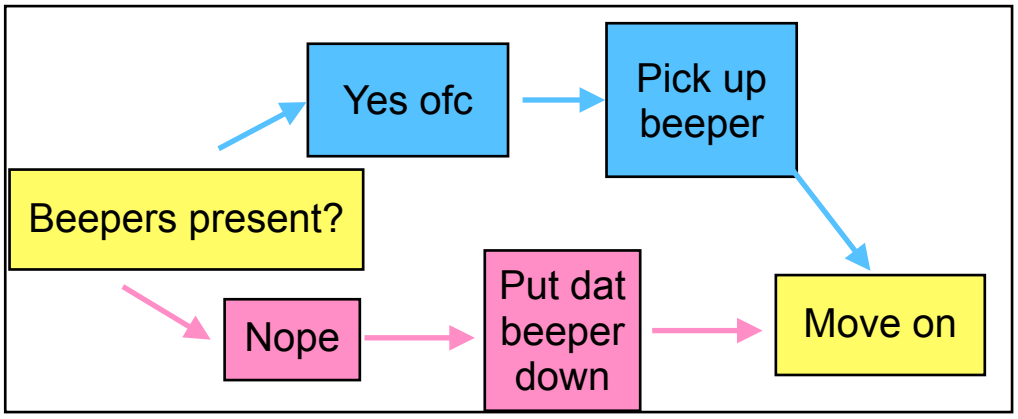
- **`if`** statements are like “siblings” to **`while`** loops because they both check for a condition to run. *However*, the key difference is that `if` statements will only execute once. (No repetition.)

```
def main():
  if
beepers_present():
    pick_beeper()
```



- **`if-else`** statements do the same thing as `if` statements, but they *specify* what should happen instead *if* the condition is not true.

```
def main():
  if
beepers_present():
    pick_beeper()
  else:
    put_beeper()
```



Notes by Regina Gerbeaux

Decomposition

WHAT IS DECOMPOSITION?

- Decomposition is **stepwise refinement**, which helps us take a BIG problem and chunk it down into smaller, more manageable (and solvable) pieces.
  - Decomposition is a *new way of thinking* — not a new coding concept!
  - Think of decomposition as taking a huge concept (like making a Thanksgiving feast) and breaking it down into much smaller steps (like cooking each dish step by step — and, you guessed it—putting it together!)
  - Synonyms for decomposition: **stepwise refinement**, **top-down** thinking
  - For more information, read the notes on **Chapter 8 from the Karel Reader**.
- **CHALLENGE PROBLEM TIME!**
  - To practice decomposition, *come up with your own cool problem for Karel to solve*.
  - Think of something you want Karel to do. It can be anything (it doesn't have to be complicated, but it should interest you.) Then try to break it down into smaller steps to achieve your goal.

BREAK IT DOWN, BREAK IT DOWN, BREAK IT DOWNNNNN (HOW TO DECOMPOSE YOUR PROBLEMS)

- You already know how to break down problems into smaller steps! How, you ask?
- Let's take, for example, your morning routine. If you were describing it to someone, you might say, "I woke up and got ready for my day before heading to work." By that definition, your `main()` code might be:

```
def main():
  get_ready()
  go_to_work()
```

- But wait! As you might have guessed already, there should be many more steps to “getting ready” and “going to work.”
- What if we *wanted* to break those steps down more? What if your friend is *more* curious about what you do to get ready?
- Tada! We’ve now taken the *higher-level* task of “getting ready” and made it smaller into steps!

```
def get_ready():
  wake_up()
  coffee()
  breakfast()
  dance_party()
  cat_memes()
```

- We could get even more detailed:

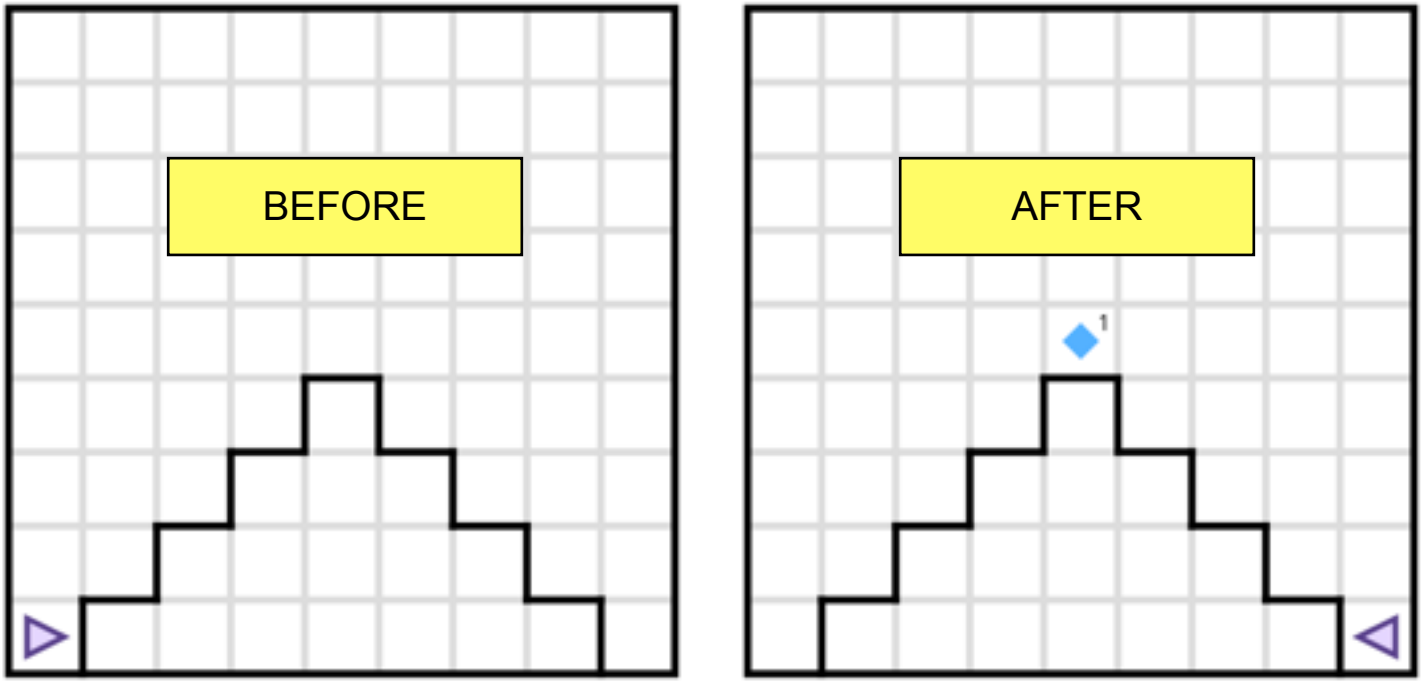
```
def dance_party():
  turn_on_spotify()
  play_taylor_swift()
  shake_it_off()
  play_queen()
  lipsync_bohemian_rhapsody()
```

- And, as you could imagine…we could get *even more* specific all the way down to each movement of your physical body (like “lift arm, shake arm around, put arm down, switch arms” etc.)
- Going all the way down to that kind of detail means we’re getting more *specific.*
- So, just as the name sounds, **top-down** approach means thinking from the **macro** perspective and dialing down to the **micro** perspective will help you break down your sections into much more manageable tasks and steps.
- If you’re curious to see these with `for` and `while` loops in action, **see the 4/15 Lecture on Control Flow** where I use coffee as an example!

Try Decomposition: Solve Problems!

Solving a Problem: Mountain Karel

- Start by trying to think of a *solution* for your problem. This would be a very general, surface-level solution (it's okay if you don't dive into all the technicalities and code jargon for now.)
- This solution is called an **algorithm** (which was invented by Muhammed ibn Musa al-Khwarizmi!)
- Here's the **problem** presented in lecture: we want Karel to scale a mountain *of any size*, plant its flag on the top, and go back down. (See image)



- Think about decomposing this into bits. Here's the **high-level solution:** Go up mountain. Place your flag. Go down mountain.
- Seems kind of simple, right? The thing is, it is. That's the *beginning* to solving harder problems!
- We'll definitely have to break these steps down a little further, though. Here's my pseudocode for the problem:

Notes by Regina Gerbeaux

Steps:

1. Ascend Mountain!

1. Karel starts facing East. Check if front is blocked.
2. WHILE it's blocked, go up mountain. To go up ONE step:
   1. Turn left (face North)
   2. Move
   3. Turn right (face East)
   4. Move
3. Once the front isn't blocked, exit loop.

2. Place a beeper at the top of the mountain.

3. We're ready to go back down!

1. Karel is facing East. It should check if front is clear.
2. WHILE it's clear, go down mountain. To go down ONE step:
   1. Move
   2. Turn right (face South)
   3. Move
   4. Turn left (face East)

High Level (bold)
Mid-Level (blue)
Low-Level (pink)

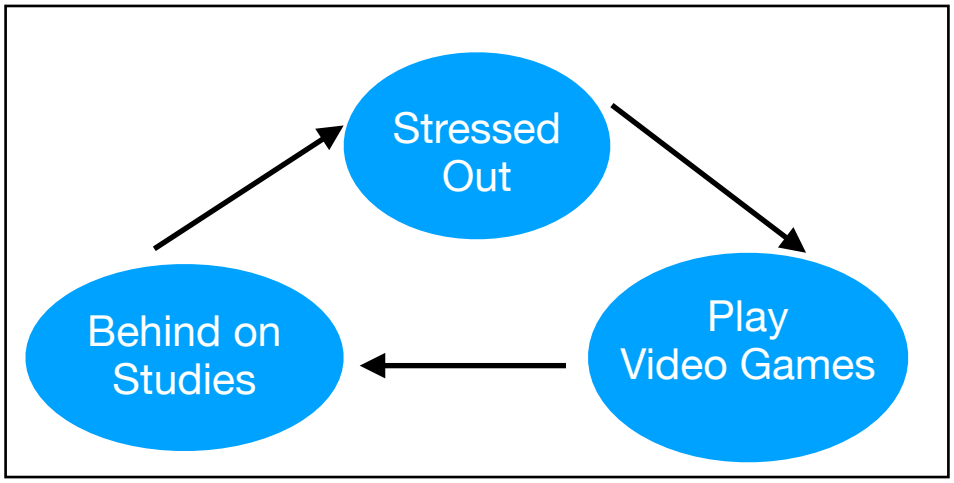
```
# go up mountain, plant the flag, go back down!
def main():
    ascend_mountain()
    put_beeper()
    descend_mountain()

# to go up mountain, check if front is blocked (Karel is facing EAST.)
def ascend_mountain():
    # while the front is blocked, Karel should step up the mountain
    while front_is_blocked():
        step_up()

# to go down the mountain, check if the front is clear (Karel is facing EAST.)
def descend_mountain():
    # while the front is clear, Karel should step down the mountain
    while front_is_clear():
        step_down()

# how Karel takes a step up the mountain!
def step_up():
    turn_left()
    move()
    turn_right()
    move()

# how Karel takes a step down the mountain!
def step_down():
    move()
    turn_right()
    move()
    turn_left()


# make Karel turn right
def turn_right():
    for i in range(3):
        turn_left()
```

TAKEAWAYS

- A good function should:
  - Do “one conceptual thing.”
  - Have clear names (see notes on *naming conventions*.)
  - Contain less than 10 lines
  - Contain 3 indentations at most
  - Be reusable and easy to modify
  - Well-commented (write what your code does!)

COMMON PITFALLS

- **Infinite loops**. Avoid writing code that has a condition that will *never* exit out, and the loop will continue forever. Yikes!
- For example, let’s say that you’re someone who **procrastinates** when you’re **stressed** out.



- That would be an example of a loop that is *infinite.* You can’t get out because you default to playing video games every single time you’re stressed out! (And this is you, as a result:)

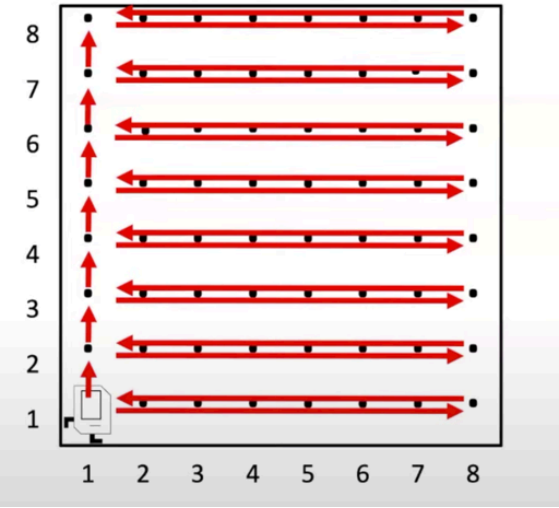
Choosing a Strategy

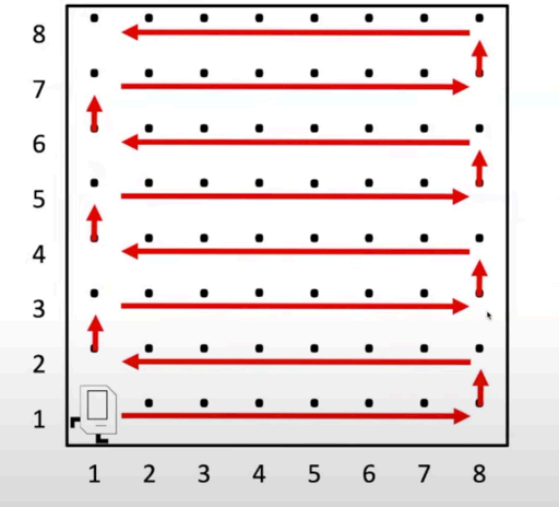
- When you're going top-down in approach to solving a problem, you want to *brainstorm* a strategy.
- As you might imagine though, there are multiple strategies that you *could* choose. So, **what is the best way to choose a strategy?**

> "Premature optimization is the root of all evil."
>
> -Don Knuth

(Had to take advantage of the opportunity to use Comic Sans MS. Sorry guys.)

  - What this quote means is, don't choose a solution just because it seems like it **has the fewest steps.**
  - Choose a solution that is likely going to **work every single time.**
- The way you choose the best strategy is to **find the solution that has the most similar precondition and postcondition.**

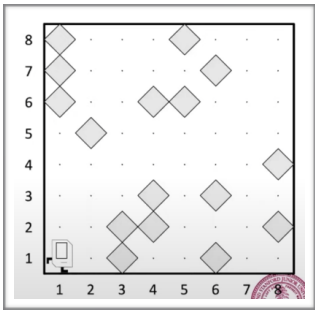




- The above example shows two possible solutions to the Roomba problem we'll be doing in a little while from now. They both accomplish the same thing: They pass through every corner in the grid. So which strategy do we choose?
  - You might be inclined to opting for the Zigzag Solution (Option B) over the Comb Solution (Option A)…however, if we go based on the **precondition and postcondition:**
    - The **Comb** Solution has a matching precondition every time (Karel always ends up at the Westmost grid, facing West.)
    - The **Zigzag** Solution has a different precondition depending on which side Karel is on (if Karel is on the Westmost grid, it's facing West. If Karel is on the Eastmost grid, it's facing East.)

The trick to choosing your strategy is this: How consistent is the precondition with the postcondition?

Solving a Problem: Roomba Karel



- **Problem**: You want Karel to become a Roomba and go through a room (of any size) to clean up all the mess! There is trash (represented by beepers) everywhere!
- There are lots of strategies you could choose to do. You could do the Spiral solution, Zigzag solution, Crazy solution, Comb solution…etc…
  - Choose the solution with the most **consistent precondition + postcondition.** So there we have it: we choose the **Comb Solution!**
- **TOP LEVEL GAMEPLAN:**
  - Go through one "line" of the room.
  - Pick up trash.
  - Go back to the start, then move one level up if there's another level.
  - Repeat until the whole room is clean!

GAME ON! Here's the solution to our problem.

Main Statement:

```
def main():
    # check if there's another row above to go to
    while left_is_clear():
        # Precondition: Karel is facing East, start of the row
        clean_one_level()
        go_back()
        # Postcondition: Karel is facing East, start of the next row
    # fencepost resolution
    clean_one_level()
```

Helper Function: `clean_one_level()`

```
# Precondition: Karel is at the start of a row filled with trash
# Postcondition: Karel is at the start of a row with no trash
def clean_one_level():
    while front_is_clear():
        clean_up()
        move()
    #fencepost resolution
    clean_up()
```

Helper Function: `clean_up()`

```
# Pick up trash if present
def clean_up():
    if beepers_present():
        pick_beeper()
```

Helper Function: `go_back()`

```
# Precondition: Karel is at the end of the row facing East
# Postcondition: Karel is at the beginning of next row facing East
def go_back():
    turn_around()
    move_to_wall()
    turn_right()
    move()
    turn_right()
```

Helper Function: `turn_around( )`

```
# turn around
def turn_around( ):
    for i in range(2):
        turn_left( )
```

Helper Function: `move_to_wall( )`

```
# Move towards the wall
def move_to_wall( ):
    while front_is_clear( ):
        move( )
```

Helper Function: `turn_right( )`

```
# turn right
def turn_right( ):
    for i in range(3):
        turn_left( )
```

And here's a fun Karel world I've created for this problem. Just copy and paste it into a new Pycharm world and save. Enjoy!

```
 Dimension: (8, 8)
BeeperBag: INFINITY
Karel: (1, 1); East
    Speed: 0.75

 Beeper: (1, 3); 1
 Beeper: (1, 4); 1
 Beeper: (1, 7); 1
 Beeper: (2, 3); 1
 Beeper: (2, 4); 1
 Beeper: (2, 7); 1
 Beeper: (3, 7); 1
 Beeper: (3, 5); 1
 Beeper: (3, 8); 1
 Beeper: (4, 1); 1
 Beeper: (4, 3); 1
 Beeper: (4, 4); 1
 Beeper: (4, 6); 1
 Beeper: (4, 7); 1
 Beeper: (5, 2); 1
 Beeper: (5, 5); 1
 Beeper: (5, 8); 1
 Beeper: (6, 1); 1
 Beeper: (6, 3); 1
 Beeper: (6, 6); 1
 Beeper: (6, 8); 1
 Beeper: (7, 2); 1
 Beeper: (7, 5); 1
 Beeper: (7, 6); 1
 Beeper: (8, 1); 1
 Beeper: (8, 4); 1
 Beeper: (8, 6); 1
 Beeper: (8, 8); 1
```

CS106A Notes - Python

Week 2 Notes: Variables in Python (4/20/2020 Lecture)

Recap on 4/17/2020 Lecture

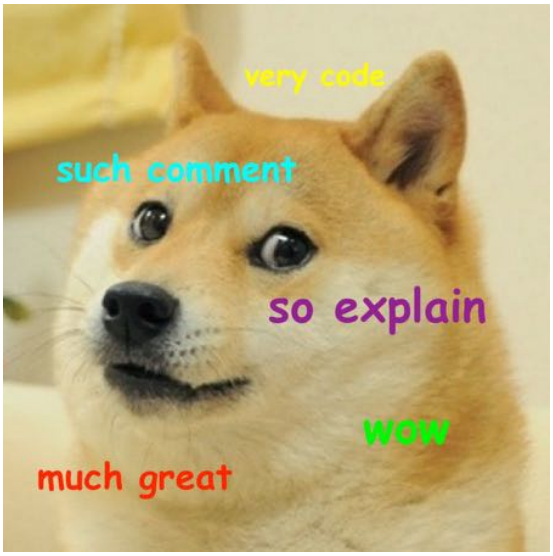
- **Programming style**:
  - Make sure your code is **well-commented**. There should be comments to describe the program and for *every* function.

```
def main():
  for i in range(8):
    if front_is_clear():
      move()
    else:
      jump_hurdle()

def jump_hurdle():
  ascend_hurdle()
  move()
  descend_hurdle()
```

This doesn't tell us anything about what we're solving, or what's going on. This is bad programming style.

```
"""
File: SteepleChaseKarel.py
--------------------------
Karel runs a steeple chase that is 9 aves long.
Hurdles are of arbitrary height and placement.
"""

# Karel needs to move forward/jump hurdles 8 times
because the race is 9 aves long.
def main():
  for i in range(8):
    if front_is_clear():
      move()
    else:
      jump_hurdle()
```

Doth my eyes deceive me? Do we have beautiful comments telling us what's happening?



Write your comments so Doge reacts like this.

- Use **descriptive names** and write them in `snake_case`.
- Functions should be **short** (1-15 lines). Anything longer than that can probably be broken into multiple (smaller) functions.

```
def function_x():
    pat()
```

Function X? What do it does? Such mystery…

```
def pet_doge():
    pat_pat_pat()
```

I like dis very much

- Keep your **indentations consistent** (Python understands what is in a loop and what is not — it is important to keep these organized.)

EACH FUNCTION SHOULD SOLVE ONE STEP OF THE PROBLEM.

- If it solves more than that, you have to break it apart (decompose it!) - this makes it easier to understand, read, and troubleshoot if you have problems in your code.

Introduction to Python: `HelloWorld`

- The first code we're going to learn how to write is a `helloworld.py` program. It is customary to have a programmer's first program write "`hello world`".

```
"""
File: helloworld.py
---------------------------
Prints the phrase, "hello, world!" to Terminal.
"""

# Our function prints the phrase, "hello, world!"
def main():
    print("hello, world!")

# this is required at the end of a Python file
# it tells program to start at the main() function
if __name__ == '__main__':
    main()
```

`print` tells the program to `print` a phrase into the Terminal.

Tells the Terminal to start running the program at `main()`.

- **How to run program in Terminal:**
  - Open Terminal in PyCharm, and type `python3 name-of-file-goes-here.py` for Mac, or `py name-of-file-goes-here.py` for PC.
    - So, for example, if you wanted to run the `helloworld.py` file from above, you would type:
      - `python3 helloworld.py` for Mac
      - `py helloworld.py` for PC
  - When you run that, you'll find the printed function directly in your Terminal!

More Complex Program: `Add2Numbers`

- This is our program, `add2numbers.py`. We'll go through it step by step together below, but let's look at what the full code looks like first.

```
"""
File: Add2Numbers.py
----------------------------
Takes two numbers from a user and adds them.
"""

def main():
    print("This program adds two numbers.")
    num1 = input("Enter first number: ")
    num1 = int(num1)
    num2 = input("Enter second number: ")
    num2 = int(num2)
    total = num1 + num2
    print("The total is " + str(total) + ".")
```

- Now we're ready to break it down step by step!

```
def main():
    print("This program adds two numbers.")
```

- This will print the phrase (a ***string***) on the Terminal, "`This program adds two numbers.`"

```
    num1 = input("Enter first number: ")
```

- An `input` is a function in Python that allows user to enter information. In this case, the Terminal will print, `Enter first number:` and accept an `input`.
- That `input` is taken from the user as a string and stored in `num1`.
  - `num1` is what we call a **variable**. Think of a variable like a **box that stores a value.** The program has now taken the user's `input` and stored it in the `num1` box.
  - For example, if a user types `9` as their `input`, `num1` now has the value `"9"`.
- **Note that the input that the user gives to the Terminal is given as a <u>string</u>** (or text, signified by double quotation marks `""`.) **It is <u>not</u> stored as an integer.**

```
    num1 = int(num1)
```

- Therefore, the next step is to **<u>create an integer version</u>** of the `num1` value (currently stored as a string.)
- We use `int()` to take the current value from `num1` and create the *integer version* of that string.
  - So now, instead of the `num1` value being `"9"` (in string format), the user input is `9` (in integer format).
  - We update the value of `num1` with the integer version of `9`.

```
num2 = input("Enter second number: ")
num2 = int(num2)
```

- The same thing from above applies to the variable `num2`. The Terminal accepts user input for the second number, takes the string, and creates the integer version to store as the value.
- Let's pretend the user input for `num2` is `"17"`. The terminal accepts that input, creates the integer version `17` from the string version `"17"`, and assigns the int `17` as the value of `num2`.

```
total = num1 + num2
```

- The variable `total` is the sum of `num1` and `num2`. (Python understands basic mathematical operations, like add, subtract, multiply, divide, and modulo.)
  - In this case, the variable `total` is the sum of the integers `9` and `17` (so the value `26` will be stored to `total`.)

```
print("The total is " + str(total) + ".")
```

- Let's look at the `str(total)` part first. This piece of code tells the `print` function to print the variable `total`. But remember how we saved `total` as an *integer*? `str()` creates the ***string*** version.
  - Since the value of `total` is the int `26`, `str()` creates a string version of `total` **so it can print as part of the sentence**.
  - Note: The value stored in `total` is *still* an int because we did not make any changes to its value; we just used `str()` to create a string version of the value for printing purposes.
  - Note: Additionally, `print` can print numbers; we only have to convert if we're concatenating other strings with them.

- Great! Let's move back to the `print` function. As a reminder, the `print` function will print the string to the Terminal.
  - Notice the addition signs `+`. We call this process **concatenation**, where you can splice together multiple strings into one full string. This allows us to add information that changes as the input changes (called **dynamic** information.)
  - Therefore, the information that will be processed is:

```
print("The total is " + "26" + ".")
```

  - Since we've concatenated the sentence, the Terminal will print, "`The total is 26.`"

Notes by Regina Gerbeaux

Functions

PRINT FUNCTION

- The `print` function prints text to the Terminal.
- `print` will print the text that is between **double quotes**, or **single quotes**. These are called **delimiters**. **Which one you choose to use depends on the kind of text you're printing.**

```
print("Oh no you didn't!")
# output: Oh no you didn't!
```

Notice how the text contains an apostrophe. You use **double quotation marks** so the Terminal doesn't interpret the apostrophe as a single quotation mark.

```
print('She said, "Hold my beer, Karel" and ran.')
# output: She said, "Hold my beer, Karel" and ran.
```

This text contains quotes. You use **single quotation marks** so the Terminal understands to print the double quotation marks as part of the text output.

- Additionally, you can **concatenate** (add) strings together in `print`.

```
str1 = "hi"
str2 = " "
str3 = "there"
```

```
print(str1 + str2 + str3)

# output: hi there
```

INPUT FUNCTION

- The `input` function gets **text** input from the user.
- `input` will `print` whatever is inside the quotes, and will wait for the user to provide input.

```
num1 = input("Enter first number: ")
```

The terminal `prints`, "`Enter first number: `" and will wait for the user to provide **text**. That text will be stored in the **variable** `num1`.

- After the user enters their input and hits **enter**, the input is stored into a **variable.**

Variables

INTRO TO VARIABLES

- A **variable** is something that stores information in a program. Think of it like a **box**. The box has a name, and there's a value stored in it.
- **Values** are **assigned** to **variable** **names**.

This turquoise box is a *variable*. Its name is `num1` and the value it stores is `10`.

- **TO CREATE A VARIABLE:** type the name, use an **equal** sign, then type the value. First time you **assign** a value to a var, you create it.

```
num1 = 10
```

Boom! You just created a variable named `num1` *with a value of* `10`*.*

- **TO UPDATE A VARIABLE:** You can also update variables. Type the name of the variable you want to update, use an equal sign, and **reassign** that variable with the new value. Any time after the first time assigning value to a variable is considered an update.
- So, for example:

```
num1 = 5
```

You just updated `num1`*'s value to* `5`*.*

- Note: **assignment is <u>not</u> the same as "equals to" in math.**
  Assignment…
  - **evaluates** the value to the *right* of the equal sign
  - **assigns** it to the variable on the *left* of the equal sign
- So, for example:

```
def main():
    total = 10
    total = total + 1
```

Step 1
The variable `total` is created with a value of `10`.

Step 2
Since **the <u>right</u> side gets evaluated first**, the value is the *current* value of total (`10`) added to 1.

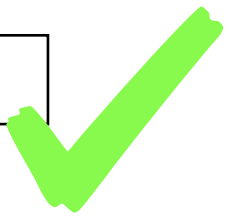
Step 3
Only **after** we've evaluated the right side, <u>then</u> we **assign** the value from the right to the *variable name* on the left. `total` *now* holds the value `11`.

- Final note: variables are only visible within the function that it lives in. (*This is also known as a **scope**.*)
  - So, if you created the variable `total` in the `main()` function, you can't call it in another function outside of `main()` — if you tried calling it in a functioned outside called `pizza()`, that function wouldn't know what you were referencing.

NAME DAT VARIABLE (VARIABLE NAMES)

- Variable names have to:
  - **Begin** with **letter or underscore**
  - Contain **only letters, digits, or underscores**
  - **Can't be a "built-in" command** in Python (e.g.: `for`)

```
good_variable_name_example
```



```
for!
```

- Variable names are **case sensitive**. (`Total` is not the same variable as `total`.)
- They should be **descriptive**, just like your function naming conventions are.
- Use **`snake_case`**.

PYTHON OBJECTS (SUITCASE ANALOGY)

- When you store information in a variable, it becomes a Python **object**.
  - Objects can come in different sizes and types.
  - Think of an object as a *suitcase* that is stored in your computer.
    - Your objects will take up computer memory (RAM)
    - Your computer has the ability to store millions of objects.
  - Each *suitcase* (holding a value) has a *luggage tag* (with the name.)

```
num_students = 700
num_in_class = 550
num_absent = num_students - num_in_class
```

Suitcase stores the value `700` and has a luggage tag that reads `num_students`.

Suitcase stores the value `550` and has a luggage tag that reads `num_in_class`.

Suitcase stores the value from `num_students` subtracted by the value in `num_in_class`. The luggage tag reads `num_absent`.

VALUE TYPES

- All variables know what *type* of information it contains.
  - For instance, in the example given above, Python recognizes that the value `700` is an *integer* type.
- Similarly, if we had a variable that stored the phrase, `“Jiminy Cricket!”`, that phrase (or any text) is a *string* type.
- Here are the different types of data available:

- `int` (integer - no decimal points)
- `float` (number with decimal point)
- `string` (text, encapsulated in either “” or ‘’)
- `bool` (boolean, True or False)

NUMBER TYPES: DIFFERENCE BETWEEN INT AND FLOAT

- What’s the difference between an `int` and a `float`?
  - `float` is a *real value number* with no next number.
    - i.e.: if I have a variable that stores a value of 1.0, we can’t determine what the next number is (it could be 2.0, it could be 1.1, it could be 1.01…)
    - Examples using floats: weight, score/grade, racing time
  - `int` is a *real value number* that does have a next number.
    - i.e.: if I have variable called `age` that stores the value 37, the next number would be 38. We can determine that next number.
    - Examples using integers: age, number of children, number of pets…

CS106A Notes - Python

Week 2 Notes: Expressions in Python (4/22/2020 Lecture)

Recap on 4/20/2020 Lecture

- In the previous lecture, we covered **converting** types in our `add2numbers.py` program from *string* to *integer.*
- Recall that we converted a string to an integer in two steps:

```
def main():
    num1 = input("Enter first number: ")
    num1 = int(num1)
```

We declared variable `num1` and set it to receive a *string* input from user.

Then, we updated the value of `num1` to convert the *string* into an *integer*.

- Here's how we can do it in one step:

```
def main():
    num1 = int(input("Enter first number: "))
```

Here, we've done it in ONE step! Just like math expressions, we've taken the value of `input` and called the `int` conversion on that value.

Intro to Expressions

INTRODUCTION TO ARITHMETIC EXPRESSIONS

- What if we want to do *more* than just addition though? (Hint: you can.)

- Python recognizes the following expressions:
  - Addition (`+`)
  - Subtraction (`-`)
  - Multiplication (`*`)
  - Division (`/`) - gives you a *float* back if number divides unevenly (e.g.: `5/2=2.5`)
  - Integer Division (`//`) - gives you an *integer* back if number divides unevenly (e.g.: `5/2=2`)
  - Remainder - or *modulo* (`%`)
    - Further explanation on remainder: if a number divides unevenly into another number, there's a *remainder*. That remainder is also known as the modulo.

    - If a number divides evenly into another number, the modulo is 0.
  - Exponentiation (`**`) - gives you the raised number of one number to the second number
    - e.g.: if we have `num1 ** num2`, that is `num1` raised to the power of `num2`.
  - Negation - or *unary* (`-`) - something is equal to the *negated* version of another number.
- Here's a summary of all the operators we just covered.

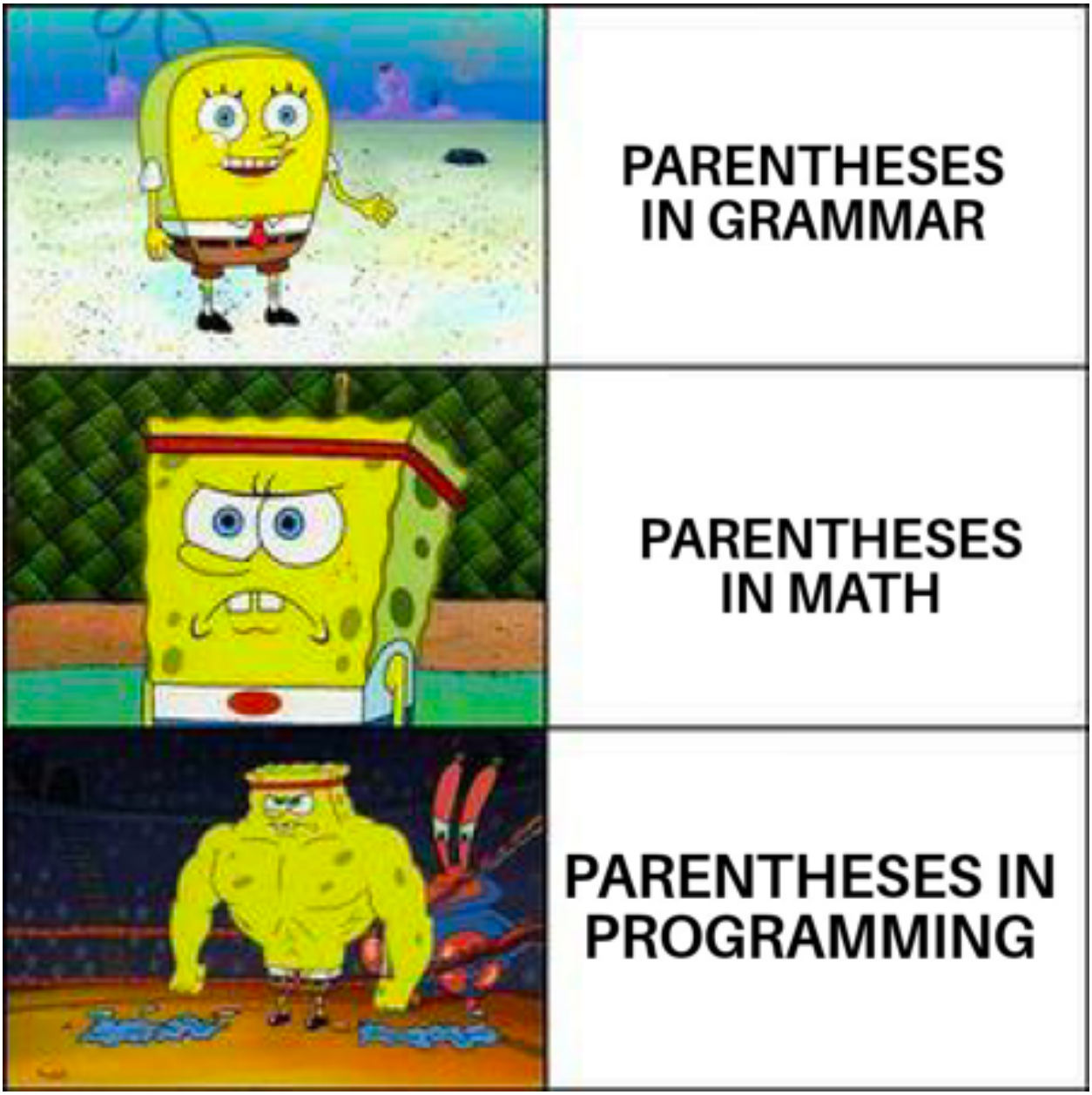
```
num1 = 5
num2 = 2
```

```
num3 = num1 + num2 # output: 7
num3 = num1 - num2 # output: 3
num3 = num1 * num2 # output: 10
num3 = num1 / num2 # output: 2.5
num3 = num1 // num2 #output: 2
num3 = num1 % num2 # output: 1
num3 = num1 ** num2 # output: 25
num3 = -num1        # output: -5
```

PRECEDENCE

- There's a hierarchy of the order that operators happen in Python, similar to PEMDAS in mathematics. Here it is, in order from highest precedence to lowest precedence:
  - Parentheses `( )` - highest (with inside parentheses computed first, then outer parentheses)
  - Exponentiation `**`
  - Unary negation `-`
  - Multiplication, division, integer division, remainder `* / // %`
  - Addition, subtraction `+ -` - lowest
- Python evaluates things that have the same precedence from left to right, just like algebra

WHEN IN DOUBT, USE PARENTHESES!



- Here's an example of bad programming style:

```
x = 1 + 3 * 5 / 2
x = 1 + 15 / 2
x = 1 + 7.5
x = 8.5
```

Bad style — you're counting on programmers to memorize order!
Here's the problem *without* parentheses.

- Here's the same example, using our friend parentheses!

```
x = (1 + ((3 * 5) / 2))
x = (1 + (15 / 2))
x = (1 + 7.5)
x = 8.5
```

Now you're specifying what order you want operations done in. Here, we go from inner parentheses all the way to outside parentheses.

IMPLICIT TYPE CONVERSION

- Sometimes, Python does conversions for you.
- For example, if you're doing an operation on two `int` (whole numbers), the result is also an `int`.

```
num1 = 5
num2 = 2
num3 = 1.9
```

```
num1 + num2 = 7
```

o hello there, int

- **The exception is division** - *This always produces a float!*

```
6 / 2 = 3.0
```

wao, float!

- If you're doing an operation on a `float`, you get back a float!

```
num3 + 1 = 2.9
```

num3 is a float, so our result is a float!

- Exponentiation depends on the result:

```
num2 ** 3 = 8
```

2 to the 3rd power is 8, which is an int

EXPLICIT TYPE CONVERSION

- You can also tell Python what *type* you want back! (Recall in our `add2numbers.py` program, we specified the conversion to `int`):

```
def main():
    num1 = int(input("Enter first number: "))
```

Using float(*value*) to create a new real-valued number:

```
num1 = 5
num2 = 2
num3 = 1.9
```

- You could also use **conversions** to create a new real-valued number. For example:

```
float(num1) = 5.0
```

we converted `num1` from an `int` to a `float`!

- **Note: the value of `num1`, an `int`, has not changed.** We are creating a new value as a `float`.
  - If we were to print `num1` even after doing this conversion, `num1` would still print as an `int` type.

```
num1 + float(num2) = 7.0
```

Since we're adding a `float`, our result is a `float`.

```
num1 + num2 = 7
```

Still, the underlying value of `num2` has not changed, so the result is an `int`.

  - If you wanted to use the `float` type after converting, you'd have to **store it into a variable**.
- Python has a function called **type** that will tell you the type of a variable if you want to find out. `type(var goes here)`
- Python evaluates types by:
  - Looking for a decimal point. If there's a decimal point, it's a `float`; if not, it's an `int`.
  - Looking for quotation marks. If there's quotation marks, it's a `string`.

Using int(*value*) to create a new integer-valued number:

- We use conversions to **truncate** anything after a decimal point. For example:

```
int(num3) = 1
```

we took `num3` and got rid of everything after the decimal point!

```
int(-2.7) = -2
```

we chopped off the decimal point just the same with negative nums!

- **Floats are not always exact.** Python will do what it can to get us *as close to* the real number as possible, but because of the floating point standard, some real numbers can't be represented exactly.

```
num1 = 5
num2 = 2
num3 = 1.9
```

```
num3 - 1 = 0.8999
```

The **type** of result is a `float`. The **value** of result is `0.8999999999999999999`

EXPRESSION SHORTHANDS

General Rule: If you have a variable and you want to write an **expression** that says that variable is equal to *itself* with an operator, the formula is to write **`variable` *`operator`*`= (expression)`**

Longhand Version	Shorthand Version
`num1 = num1 + 1`	`num1 += 1`
`num1 = num1 - 1`	`num1 -= 1`
`num1 = num1 * 1`	`num1 *= 1`
`num1 = num1 / 1`	`num1 /= 1`

Problem Solving: `average2numbers.py`

- Goal of the program: to ask a user for two numbers and print the average of those two numbers.

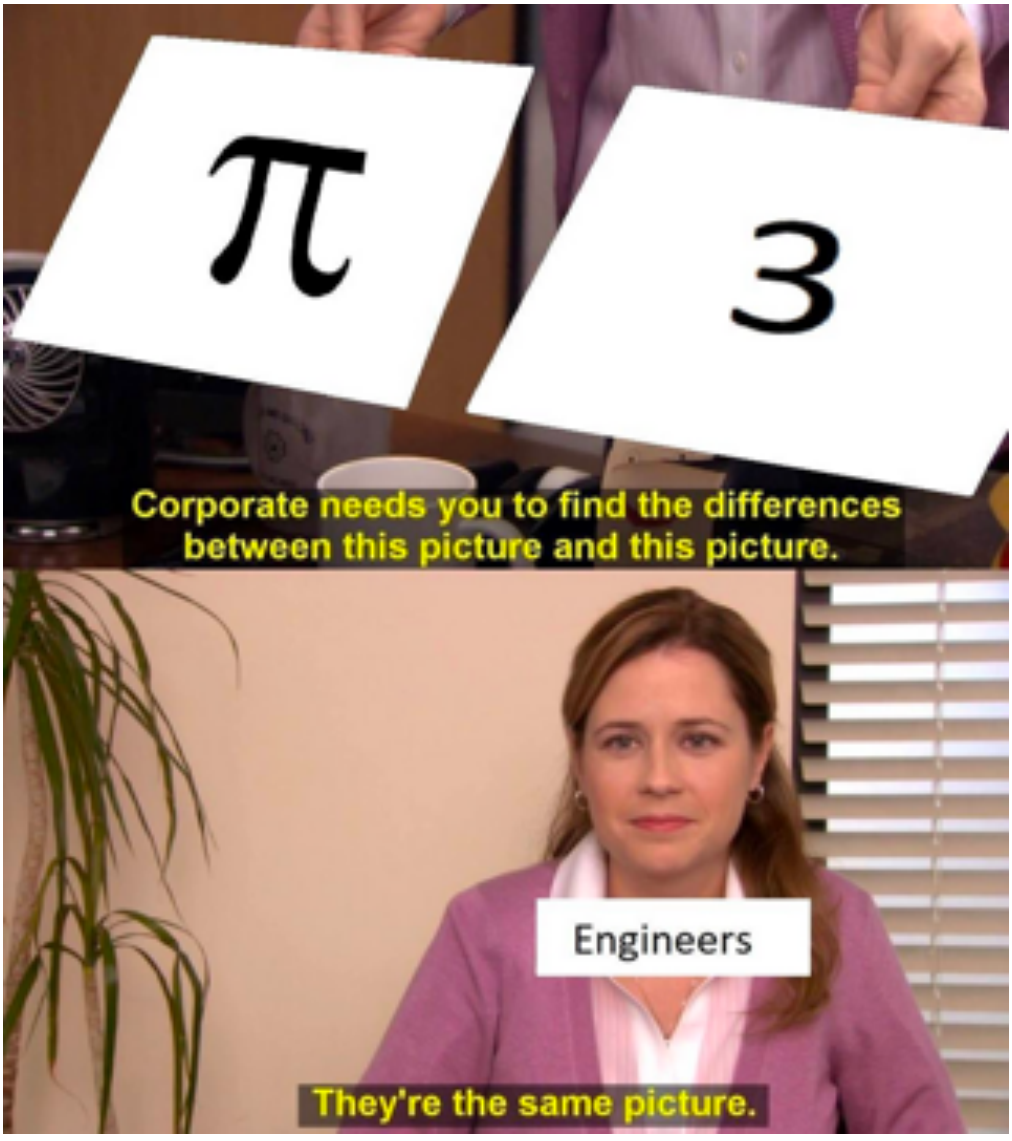
```
def main():
  print("This program averages two numbers.")
  num1 = float(input("Enter first number: ")
  num2 = float(input("enter second number: ")
  total = (num1 + num2) / 2
  print("The average is " + str(total) + ".")
```

- Notice we convert the user input into floats in order to correctly compute the average.
- Notice the use of parentheses so that Python knows what order to perform the calculations.

Intro to Constants

INTRODUCTION TO CONSTANTS

- `constants` are pieces of code you will reuse throughout your code! They make your code easier to read and follow good style.
- `constants` do not change while we're running the program (but you can change it *between* runs, for better precision, for example)
- Take the following example: `area = 3.1415 * (radius ** 2)`
- Although it isn't necessarily ugly, it would be helpful to trace back `3.1415` to the value of Pi. Imagine if we had to use Pi throughout our code! This is where saving a constant helps.



- Saving the value to a constant gives us cleaner code, and also helps others reading your code understand the operation better.

```
PI = 3.1415

area = PI * (radius ** 2)
```

- The above code clearly communicates to others reading your code that the formula for calculating `area` is `PI` multiplied by `radius squared`.
- As mentioned above, we *can* update constants between program runs. Let's say we needed to make `PI` more accurate. Rather than going through our *entire* program manually updating every single occurrence of `3.1415`, we go straight to the `constant` declared and update that just once, to `3.141592653589793`. Now, everything in your code that calls `PI` has been updated!
- **Naming convention** for `constants`:
  - `WRITE THE NAME OF THE CONSTANT IN CAPS`.
  - Use `SNAKE_CASE`.
  - Use descriptive names.
- **Code should be written with constants in a general way so that it can still work if constants are changed.**

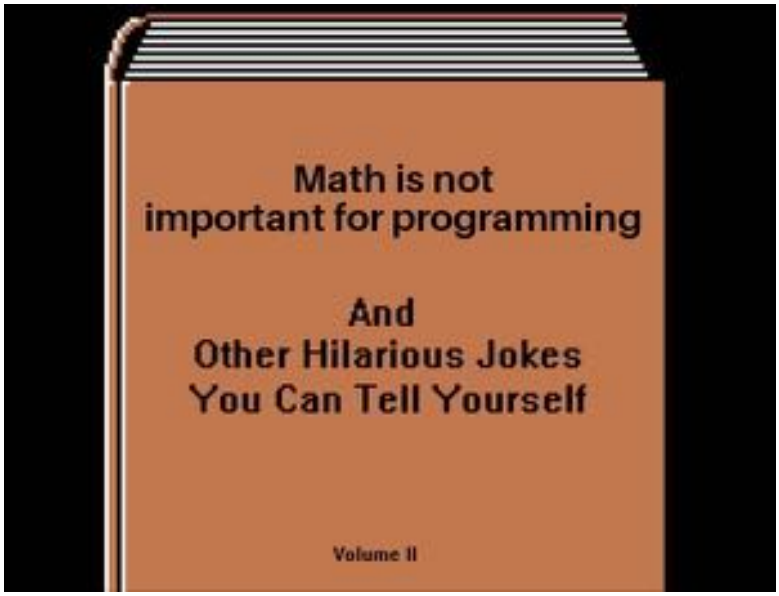
Problem Solving: `constants.py`

```
INCHES_IN_FOOT = 12

def main():
  feet = float(input("Enter number of feet: "))
  inches = feet * INCHES_IN_FOOT
  print("That is " + str(inches) + "inches!")
```

- In this above code, we have a nice example of using a `constant`.
- Notice the `constant` `INCHES_IN_FOOT` is declared *before* the main function. It lives in the entire program!
- First, we take the user input and convert it into a `float`.
- Next, we declare the variable `inches` and perform the mathematical operation of `feet` multiplied by the constant `INCHES_IN_FOOT`
- Finally, we convert `inches` into a string in order to print the full text!

The Python `math` Library

- There is a **library** available in Python called `math`.
- In order to use a library, you have to use an `import` statement!
- The `math` library contains many `constants` that are already built in, including `math.pi` and `math.e`



- The `math` library also contains many `functions`, such as `math.sqrt(x)` which returns the square root of x.
- **You can look up the full list of `math` operations by searching "Python math library" in Python3 documentation.**
- To use a constant or function inside the `math` library, we prepend our statement with `math.`*`name_of_function`*.
- From our Pi example above, we could do this instead:

```
import math

def main():
  area = math.pi * (radius ** 2)
```

- Also, notice that we removed the constant declaration now and instead imported the entire `math` library!
- Notice how we prepend the pi constant with `math.`

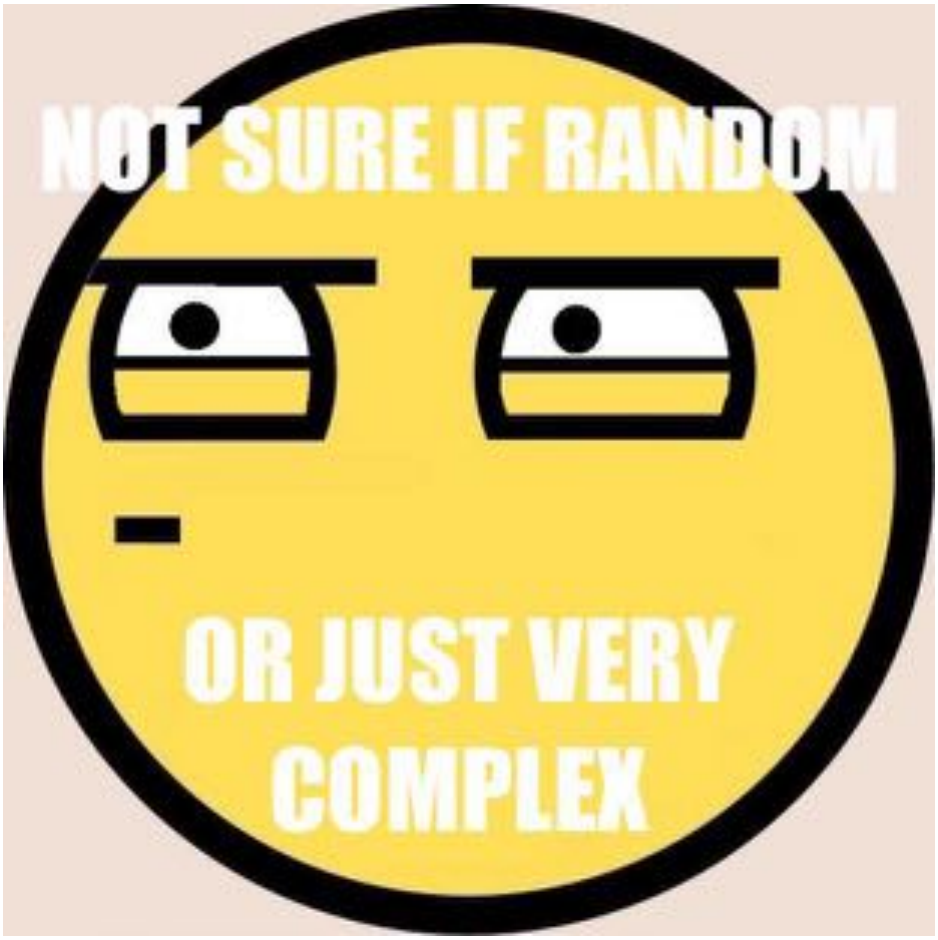
`math` Example: `squareroot.py`

- Here's an example of using the `math` library in our code. This is our program `squareroot.py`.
- The purpose of this program is to compute square roots. Cool!

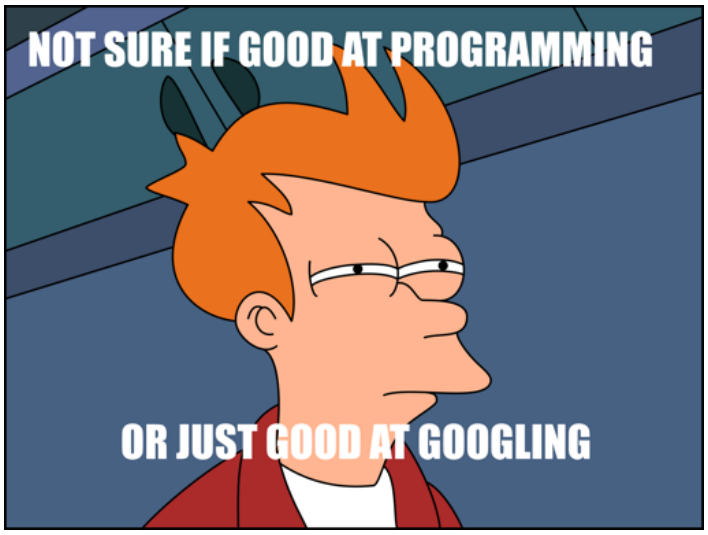
```
import math

def main():
  num = float(input("Enter number: "))
  root = math.sqrt(num)
  print("Square root of " + str(num) + "is " + str(root))
```

- First, we have an `import math` statement to give our code access to all of the `math` constants and functions
- Then, in our `main` function, we declare the variable `num` and set it to take input from the user. The input is converted into a `float`.
- We declare another variable called `root`, and assign it the value of `math.sqrt(num)`. Here, we're telling the program, "The value of `root` = the function `math.sqrt` on our variable num!"
- The program will now print the statement, "Square root of *stringified number goes here* is *stringified root goes here*."

Random # Generator: `random` library



- What if we want to generate a random number, for example, to use in a game?
- We can use a function to generate a *sequence* of *pseudorandom* numbers!
  - Note: the sequence is pseudorandom because a computer can't *actually* produce random numbers, but it *does* appear random to us humans.
- Additionally, we can tell the computer to always start with a particular path, thus producing the same **sequence**. This is called **seeding** the random number generator.



- A **seed** is telling the random number generator to always start with and produce *that* same sequence.
- **What's the point if generating the same sequence every time? That's not very random!**
- If your program is having issues and you're trying to figure out *what* those issues are while debugging, it would be nice to have the same sequence come back each time so you know where your program goes incorrectly.
- Once your program works correctly, you can *remove* the seed so it's truly "random" every time!

- Just like the `math` library, there's a bunch of `random` functions you get access to when you `import` it. This includes:

FUNCTION	WHAT IT DOES
`random.randint(`*`min`*`, `*`max`*`)`	Returns a random int between *min* and *max*, inclusive
`random.random()`	Returns a random real number (float) between 0 and 1
`random.uniform(`*`min`*`, `*`max`*`)`	Returns a random real number (float) between *min* and *max*
`random.seed(`*`x`*`)`	Sets "seed" of random number generator to *x*.

Problem Solving: `rolldice.py`

Let's get `rolldice.py` to work! The purpose of this program is to generate two random dice rolls, and then to add up the total of the dice values.

```
import random

NUM_SIDES = 6

def main():
  die1 = random.randint(1, NUM_SIDES)
  die2 = random.randint(1, NUM_SIDES)
  total = die1 + die2
  print("Dice have " + str(NUM_SIDES) + "sides each.")
  print("First die: " + str(die1))
  print("Second die: " + str(die2))
  print("Total of two dice: " + str(total))
```

- First, we import the `random` library.
- Then, we declare the constant `NUM_SIDES` and assign it the value 6. **If we wanted to change the number of sides our dice have, we would update it here and it would affect the rest of code.**
- Next, in our main function, we create two variables, `die1` and `die2`.
  - Both are assigned the value of the `random.randint()` function, which will generate a number between 1 and the value of `NUM_SIDES` (which is currently 6.)
- Then, we create the variable `total`, which takes the numbers from `die1` and `die2` and adds them together.
- Finally, we have a lot of `print` statements!
  - First, the Terminal will print, "`Dice have` *`stringified value of NUM_SIDES`* `side each.`"
  - Then, it will print, "`First die:` *`stringified value of die_1`*" and then, "`Second die:` *`stringified value of die_2`*"
  - Finally, it will print, "`Total of two dice:` *`stringified value of total`*"
- Below is an example of what the Terminal could print:

```
Dice have 6 sides each.
First die: 4
Second die: 2
Total of two dice: 6
```

Debugging Our Code: Set a Seed!

- Remember how setting seeds can help us debug? We can do so by adding `random.seed(1)` to our code in the `main()` function. What this does is ensure that the path generated will always be `1`. (Note:

This does *not* mean the output number will be `1`; it just means that it will always produce the same generated sequence.)

- Adding this to our code, it would look like this:

```
import random

NUM_SIDES = 6

def main():
  random.seed(1)
  die1 = random.randint(1, NUM_SIDES)
  die2 = randomrandint(1, NUM_SIDES)
  total = die1 + die2
  print("Dice have " + str(NUM_SIDES) + "sides each.")
  print("First die: " + str(die1))
  print("Second die: " + str(die2))
  print("Total of two dice: " + str(total))
```

- This would result in the same sequence being generated every single time! This is good for debugging our code. When you no longer want it to generate the same sequence over and over again, simply remove the `random.seed(1)`.

Problem Solving: `dicesimulator.py`

The purpose of `dicesimulator.py` is to illustrate the differences between variables within different functions. See below for the code!

```
import random

NUM_SIDES = 6

def roll_dice():
  die1 = random.randint(1, NUM_SIDES)
  die2 = randomrandint(1, NUM_SIDES)
  total = die1 + die2
  print("Total of two dice: " + str(total))

def main():
  die1 = 10
  print("die1 in main() starts as: " + str(die1))
  roll_dice()
  roll_dice()
  roll_dice()

  print("die1 in main() is: " + str(die1))
```

We define a *new* function called `roll_dice()`!

This `die1` is not the same as `die1` in `main()`

Notice we're defining the value of `die1` in `main()` as `10`.

`roll_dice()` is called 3x! 😱

Prints the val of `die1` in `main()`

This is what could get printed in the Terminal.

```
die1 in main() starts as 10.
Total of two dice: 4
Total of two dice: 7
Total of two dice: 3
die1 in main() is: 10
```

This is part of `main()`.

These are part of `roll_dice()` which have nothing to do with `main()`.

This is part of `main()`.

Let's go through this step by step, shall we?

```
def main():
  die1 = 10
  print("die1 in main() starts as: " + str(die1))
```

- In our main func, we have a variable called `die1` whose value is `10`.
- Remember that the variables we create live only inside the function where it was created. So this `die1` exists only in `main()`.
- This will print the following:

```
die1 in main starts as: 10
```

- Next, we'll exit out of `main()` and go into our `roll_dice()` function.

```
def roll_dice():
  die1 = random.randint(1, NUM_SIDES)
  die2 = randomrandint(1, NUM_SIDES)
  total = die1 + die2
  print("Total of two dice: " + str(total))
```

- Next, this is our `roll_dice()` function. Notice *also* how we have a variable called `die1`, but the value is different! **Our `roll_dice()` function has no knowledge of the `die1` that lives in the `main()`.** This is *outside of the scope.*
- This might print the following:

```
Total of two dice: 6
```

- Since `roll_dice()` has nothing to do with `main()`, it generates its *own* values for the variables `die1`, `die2`, and `total`.

- Since we call `roll_dice()` three times, it happens three times, independent of each other.
- After all three of the prints in the Terminal, the function will call the last line:

```
print("die1 in main() is: " + str(die1))
```

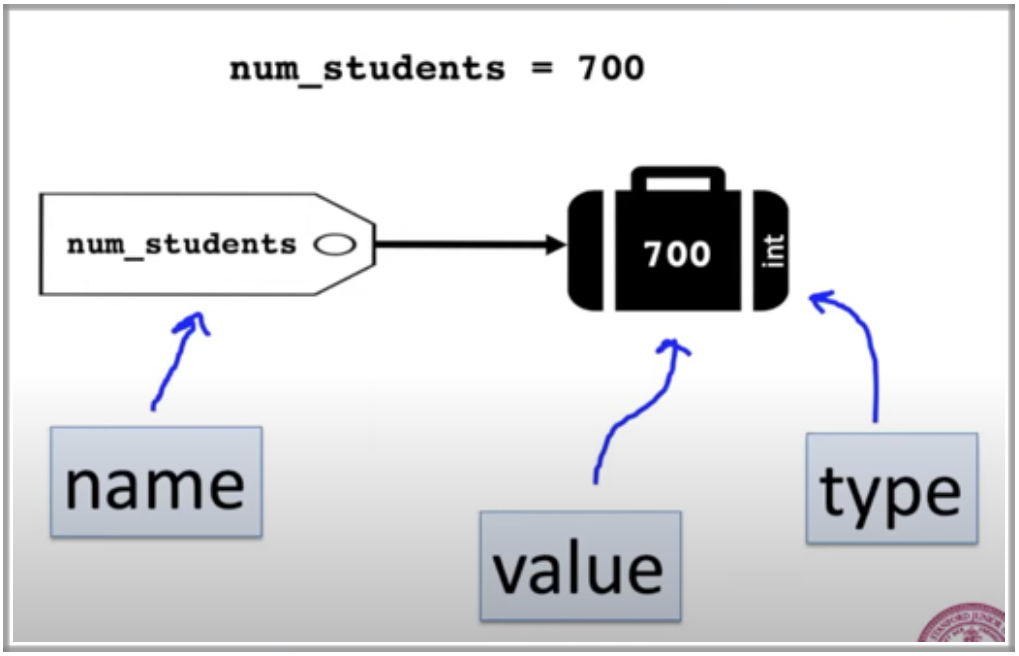
- This will print `10` still, since the value of `die1` in `main()` is unaffected by `roll_dice()`.

CS106A Notes - Python
Week 2 Notes: Control Flow Python (4/24/2020 Lecture)

**NOTE: I was in a really good mood today, so there are significantly more curse words. Since Code in Place students are 18+, I decided it would be okay to leave them in. You have been warned. :)

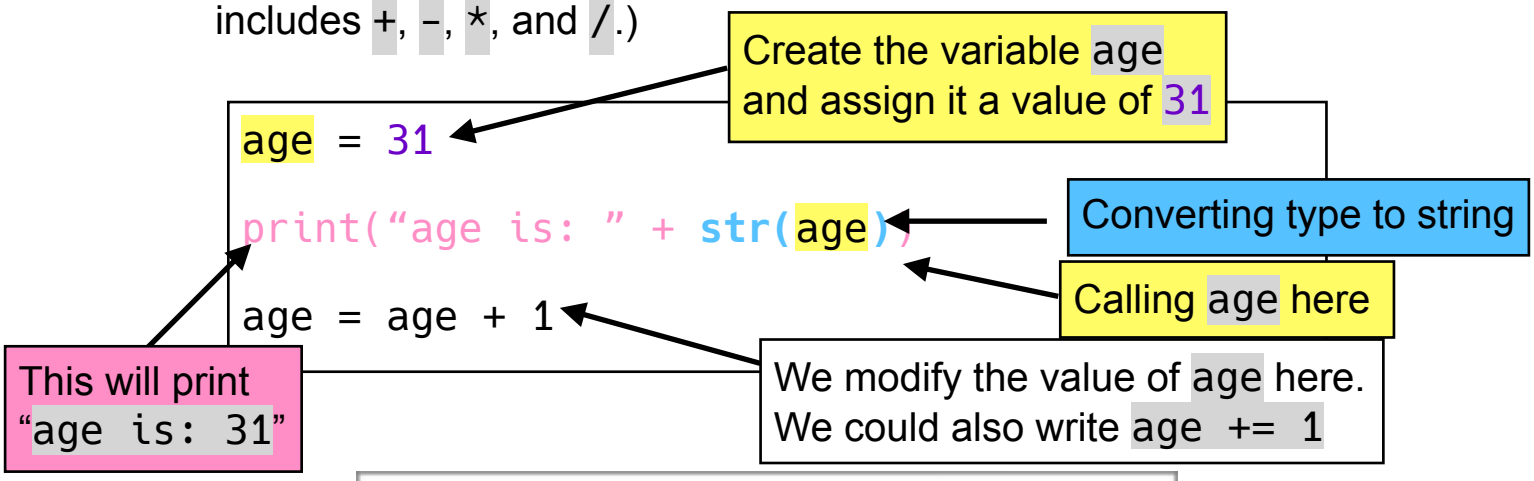
Recap on 4/22/2020 Lecture

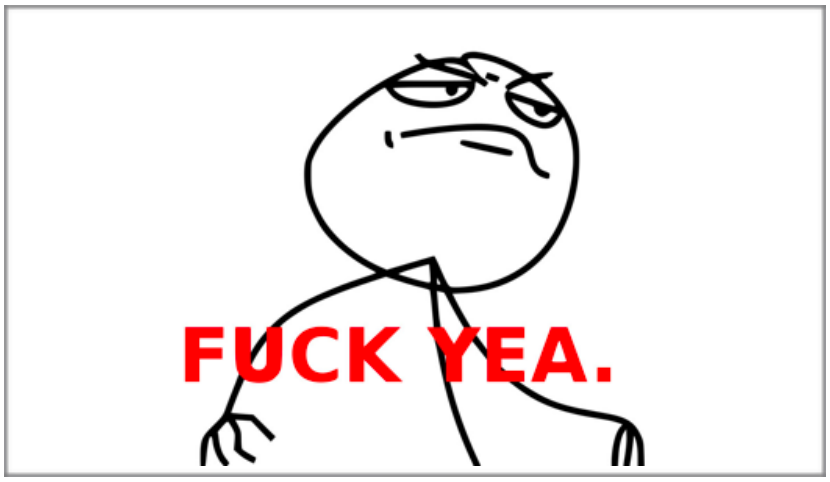
- **Variables** are how we store stuff.
- The suitcase analogy is as follows:
  - The suitcase *is* the variable. A variable has three things:
    - The suitcase has a tag that contains the **name** of the variable.
    - Inside the suitcase, it holds a **value**.
    - The suitcase knows what **type** of value it holds.
- Computers can store billions of variables. So go crazy. (But not too crazy.)



Screenshot of Suitcase Analogy taken from Lecture Recap.

- **To Create a Variable:**
  - **Name** your variable and add a **=** to **assign** your variable a **value.**
- **To Use a Variable:**
  - Call a variable by its name. (Type the variable name in your program to call it!) The computer will then:
    - Go through the list of variables you've created already
    - Find the variable with that name and grab the value
    - Bring it back to the function where we call the variable.
- **To Modify the Variable's Value:**
  - Start on the right hand side of the equal sign (value), then put it into the left hand side.
  - If you have expressions, variables, or constants, you can use mathematical operators on them to change the values. (This includes `+`, `-`, `*`, and `/`.)





Conditions

`WHILE` LOOPS AND `IF` STATEMENTS

- We've talked about **control flow** in Karel, so here we'll learn about it in core Python.
- `while` loops function **the exact same way in Python** as how we learned about them in Karel.
- The program will check if the condition is *True* or *False*. `while` the condition is true, the loop will execute. When the condition is *False*, the loop will stop.
  - Conditions evaluate to `True` or `False`. These are **Boolean**!
- The same goes for `if` statements, which function similarly to `while` loops. The difference is `if` statements evaluate as *True* or *False*, and happen only once. **Again, the difference is `if` statements can execute just once if the condition is true, whereas `while` loops will repeat as long as the condition is still true.**
- Let's take a look at our Karel code and Python code side by side:

```
while front_is_clear():
  body
```

We saw this in Karel.

```
while condition:
  body
```

This is what it looks like in Python!

```
if beepers_present():
  body
```

We saw this in Karel.

```
if condition:
  body
```

This is what it looks like in Python!

- Notice that the **conditions are different** in Python. The beauty of Python conditions is *you can write anything that evaluates to either true or false!* This makes these pieces of code very powerful!

SHOW ME A FUCKIN' EXAMPLE (`WHILE` LOOPS AND TACOS)

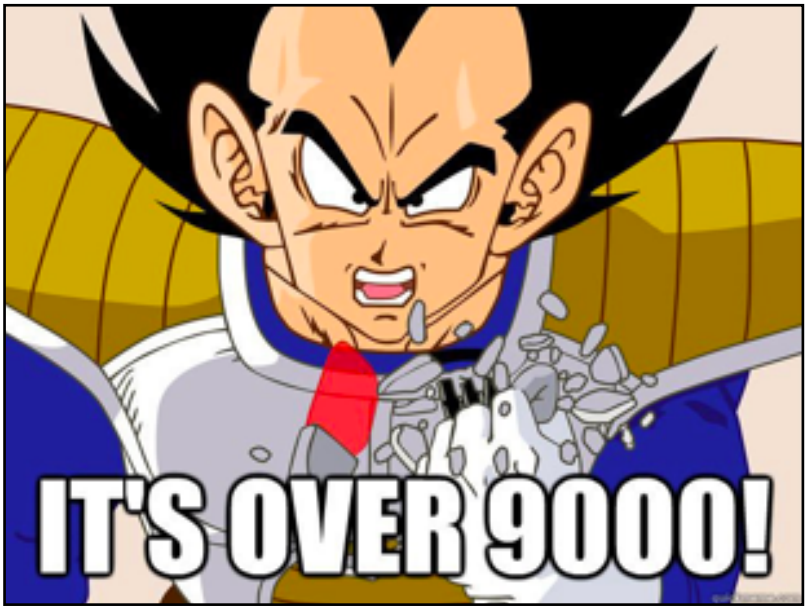
- If you wanted to write a `while` loop that never ends, we can simply write its condition to be `True`.
- This is because `True` will always evaluate to…you guessed it… `True`!
- **Example time!** Let's say I want to eat lots of tacos. So, so *so* many tacos, that I literally never stop eating tacos!
- The simplest way to do this is to write a `while` loop and to set its condition to `True`.

```
File: tacos.py
def shove_taco_in_mouth():
  print("nom")

def main():
  print("EAT DOSE TACOS")
  while True:
    shove_taco_in_mouth()
```

- Here's what my Terminal will print:

```
python3 tacos.py
EAT DOSE TACOS
nom
nom
nom
nom
nom
nom
nom
nom
nom
nom
```

`nom` prints forever…and she lived happily ever after. On a mound of tacos.



BOOLEANS

- To make the best use of `while` loops, we should learn Booleans more in-depth!
- A Boolean will always evaluate to either `True` or `False`. (Notice that we capitalize the `T` and `F`, respectively.)
- We have things called **comparison operators** that can evaluate to either `True` or `False`. Here is a list of them:

Operator	Meaning	Example	Value
==	equals	1 + 1 == 2	TRUE
!=	does not equal	3.2 != 2.5	TRUE
<	less than	10 < 5	FALSE
>	greater than	10 > 5	TRUE
<=	less than or equal to	126 <= 100	FALSE
>=	greater than or equal to	5.0 >= 5.0	TRUE

- **Let's talk a little bit more about the Comparison Operators `==` and `!=`.**
  - `==` asks, "Is the thing on my left *exactly the same* as the thing on my right?"
  - `!=` asks, "Is the thing on my left *different* from the thing on my right?"
- Here is an example of a Boolean (is 1 **less than** 2?): `1 < 2`
- Easy enough, right? The Boolean of the above expression is `True`, because indeed…1 happens to be less than 2.
- Let's see this in action *within an* `if` *statement!*

```
if 1 < 2:
  print("1 is less than 2")
```

This statement will print if the condition is True.

OkAy BuT LiKe CaN u dO sOmEtHiNg CoOLeR??

(Lol. Don't mind me. I'm having fun with this.)

```
num = int(input("Enter a number: "))
if num == 0:
  print("That number is 0.")
else:
  print("That number is not 0.")
```

THE PROGRAM CAN READ YOUR MIND. (Ok, not really—but still.)

dO sOmEtHiNg CoOLeR BrO (`if-else` statements)

- Here, we have an example where **the program will evaluate if the input is 0**. If it's not 0, the program will evaluate if the input is greater than 0 (positive) or less than 0 (negative.)

```
num = int(input("Enter a number: "))
if num == 0:
  print("That number is 0.")
else:
  if num > 0:
    print("Your number is positive!")
  else:
    print("Your number is negative.")
```

We check if the input is 0.

We check if input is greater than 0.

If `num` is not 0 *or* greater than 0, it's *implied* it is less than 0 (hence, we simply type `else`.)

INTRODUCTION TO `ELIF` STATEMENTS

But first, a very famous quote…

> "What the fuck are `elif` statements and why are there people using this in the CS106A discussion forums already???"
>
> -99% of CS106A Students

- Fret not, dear learners! We, too, shall know what `elif` statements are today. :)

- `elif` is the shorthand for `else if`. Remember the code above? There were SO many if else if else statements…confusing!
- Here is the *exact same code* as above, but reconfigured with `elif`. Notice we're not nesting `if-else` statements within other `if-else` statements here. The indentation is much nicer, and the logic is easier to read and follow.

```
num = int(input("Enter a number: "))
if num == 0:
  print("That number is 0.")
elif num > 0:
  print("Your number is positive!")
else:
  print("Your number is negative.")
```

CHECK THIS OUT. It's the same as the code above, but cleaner, thanks to `elif`!

`else` will *only* run if `if` and `elif` evaluate as False.

AND NOW YOU KNOW.

Problem Solving: Guess My Number

`GUESSMYNUMBER.PY`

- Objective of our program: to guess a number between 0 and 99 that the program has chosen at random!

- **DECOMPOSE DAT SHIT**
  - First, we need the computer to generate a random number.
  - Next, we should accept a user input for a *guess*.
  - The program should continue running until the number is guessed correctly. We can use a `while` loop.
  - The program should tell the user if the number is too high or too low in order to help the user guess the number more accurately, then prompt the user for another guess.
  - Lastly, the program should congratulate the user on making the correct guess once it's correct!

STEP 1: GENERATE RANDOM NUMBER

```
secret_number = random.randint(1, 99)
print("I am thinking of a number between 1
and 99..."
```

- First, we need our program to come up with a random number between 1 and 99!
- We declare a variable named `secret_number`.
- Its value will be a randomly generated number using `random.randint()`, which generates a random integer between 1 and 99 (inclusive!)
- The Terminal will print the phrase, "`I am thinking of a number between 1 and 99...`"

STEP 2: TAKE A GUESS FROM USER

```
guess = int(input("Enter a guess: "))
```

- Next, we need to accept a guess from our user.
- We declare a variable named `guess`.
- Its value will be an `input` from the user. The text shown to the user is `Enter a guess:` .
- We have to convert the user's guess, given back to us as a `string`, into an `int`.

STEP 3: LOOP TIME!


```
while guess != secret_number
  if guess < secret_number:
    print("Your guess is too low. Wompwomp")
  else:
    print("Your guess is too high. Wompwomp")

  print("") #creates an empty line
  guess = int(input("Enter a new guess: "))
```

- We don't want our program to exit out prematurely when the user's guess is wrong. (Chances are, the first guess won't be correct. Unless they're fucking CHEATERS. Just kidding.)
- Therefore, we can use a `while` loop to keep the program running as long as the guess is incorrect.
- `while guess != secret_number` reads as, "While the guess is different from the secret number…" which evaluates as True if the guess is *not* the secret number.

- We want the user to have a better chance of guessing the secret number, so let's tell them if their guess is too high or too low.
- The next piece of logic, an `if` statement, checks if the value of `guess` is *less than* `secret_number`. `if guess < secret_number:` reads as, "If the guess is less than the secret number…" followed by the print statement that informs the user their guess is too low. (Womp womppppp.)
- Similarly, the next piece of logic (the `else` statement) infers that if the guess isn't too low, it must be too high. Read as, "If your guess is *not* less than the secret number, it must be too high." and will print a statement that informs the user their guess is too high. (Womppp.)
- Lastly, we have to prompt the user for a guess again! Since we've declared the variable `guess` already, we are updating the variable here. The value of `guess` is now yet another user input converted into an `int`, but this time, the Terminal will print instead, "`Enter a new guess: `". Cool!
- This will run *while* the guess is incorrect. Once the guess is correct, the loop will exit!

STEP 4: FINIT

```
print("Congrats! The number was: " + str(secret_number))
```

- Hooray, the user has guessed and cursed and screamed expletives (hopefully not?) at the program, and they've *guessed the correct number!* This is because you used your awesome programming skills to let the user know whether their guesses were too low or too high. You're basically Marco Polo! ("Marco…Polo!…Marco…Polo!")
- You should let the user know they've guessed the correct number. We can use a `print` statement telling the user what the number is.

Problem Solving: Sentinel Loops

`SENTINEL.PY`

- A **sentinel** is a value that signals the end of user input.
- A **sentinel loop** is a loop that repeats until a **sentinel** value is seen.
- The example we had in lecture included the following prompt:
  Write a program that prompts the user for numbers until the user types -1, then output the total of the numbers.
  In this case, **-1** is the sentinel value because it ends the loop.

- Decomposition Time!
  - First, we can create a variable `total` to represent the total the Terminal will print out at the end.
  - Next, we create a variable that prompts the user to enter a number. Since we're planning on printing `total` at the end, we can update `total` with the user input every time a number is entered.
  - Loop should continue running until `-1` is inputted.
  - Once `-1` is entered, the loop should end and the program should print out the total of the user inputs.

STEP 1: CREATE `TOTAL`

```
def main():
  total = 0
```

- First, we create a variable called `total` and assign it the value of `0`. This is because nothing has been input by the user yet.

STEP 2: ACCEPT USER INPUT AND UPDATE `TOTAL`

```
    num = int(input("Enter a num: "))
    total += num
```

- Next, we prompt the user to type in numbers. We covert their input from string to integer.
- Now that the user has input a number, we can update the value of `total`. We do this by adding the value of `num` to the current `total`.

STEP 3: PUT THE CODE FROM STEP 2 IN A LOOP

```
  num = int(input("Enter a num: "))
while num != -1:
    total += num
    num = int(input("Enter a num: "))
```

- Since our program should keep accepting input until `-1`, we should put the code we wrote in Step 2 inside a `while` loop. We have some rearranging to do, though…
- **First, notice that we include an additional** `num = int(input("Enter a num: "))` **statement.** Why is that?

  - Since we're creating a `while` loop that uses the variable `num`, we need to create the `num` variable before the loop so the program knows what `num` is.
  - If we try running it without the pink code, the program will return an error because it doesn't know what `num` is.
  - This is very similar to the **fencepost bugs** we covered in Week 1 while working with Karel. We can accept an input first, and then start our loop, and this will prevent the program from adding `-1` to `total`.
- **Next, notice that we reverse the order of our code from Step 2.**
  - This is because we addressed the fencepost bug above! Once the user has entered a number (presumably not `-1`), we can immediately add `num` to `total`.
  - Then, we prompt the user to enter in another `num`. If it's not `-1`, the loop will run again!
- This loop will continue as long as no `num` values are `-1`.

STEP 4: PRINT TOTAL!

```
print("Your total is " + str(total))
```

- The last step once the user has typed `-1` is to print the total.
- Don't forget, we have to turn the value of `total` into a string in order to concatenate it with another string!

Here is all of the code together:

```
def main( ):
  total = 0

  num = int(input(“Enter a num: ”))
  while num != -1:
    total += num
    num = int(input(“Enter a num: ”))

  print(“Your total is ” + str(total))
```

LAST PIECE OF INFORMATION: USING BREAK

- What if we wanted a different way to end the loop above? You could write the following code instead:

```
def main():
  total = 0
  while True:
    num = int(input("Enter a number: "))
    if num == -1:
      break
    total += num
  print("Your total is " + str(total))
```

- `break` **immediately exits a loop** if the condition is met!
- In our above code, if the value of `num` is the same as `-1`, `break` is used and the loop immediately ends.
- Notice also how we updated our `while` loop so the condition is always `True`. Since we now have a `break` statement that will exit the loop for us, we can make the `while` loop conditional always true so that it continually runs (until our `break` statement is used.)

Booleans

MORE ABOUT BOOLEANS

- You can make **compound expressions** with booleans using logical operators.
- These are the three logical operators (in order of precedence):

Operator	Example	Result
`not`	`not (2 == 3)`	TRUE
`and`	`(2 == 3) and (-1 < 5)`	FALSE
`or`	`(2 == 3) or (-1 < 5)`	TRUE

`not` **takes what was False and turns it True, and takes what was True and turns it False.**

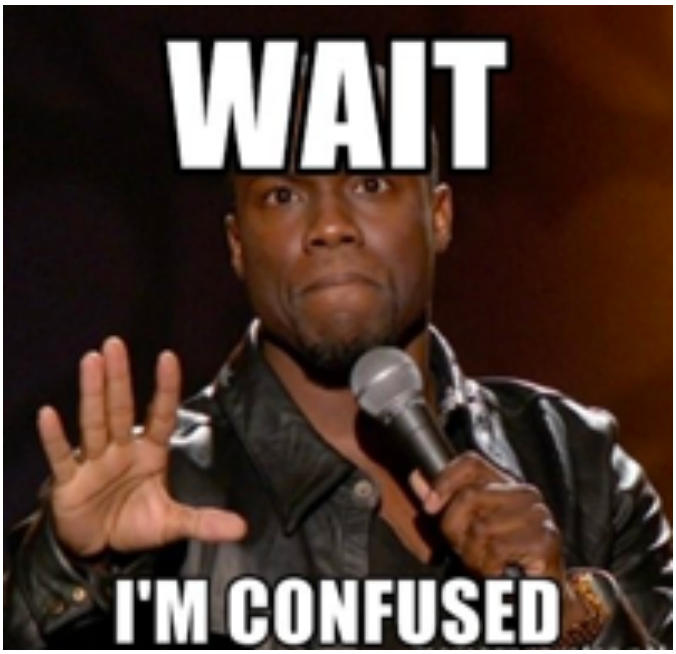
- In our above example, `not (2 == 3)` asks: "Is the following statement true? 'It is not true that 2 is equal to 3.'"
- That statement is true: "It is *not* true that 2 is equal to 3." Therefore, it evaluates as `True`.

- `and` **checks if two or more statements are true.**
  - In our above example, `(2 == 3) and (-1 < 5)` asks: "Are *both* of the following statements true?: '2 is equal to 3' and '-1 is less than 5.'"
  - Although -1 is less than 5, we know 2 is not equal to 3. Because *one* of the two statements is false, it therefore evaluates as `False`.
- `or` **check if one of two (or more) statements are true.**
  - In our above example, `(2 == 3) or (-1 < 5)` asks: "Are *one* of the following statements true?: '2 is equal to 3' or '-1 is less than 5.'"
  - With an `or` operator, only one statement has to be true for it to evaluate as `True`. Because *one* of the two statements is true, it therefore evaluates as `True`.

PRECEDENCE

- The computer will evaluate (in order of precedence) **arithmetic** followed by **comparison**, then **not**, and finally **and/or**.
- For example:

```
5 * 7 >= 3 + 5 * (7 - 1) and not False
```



- It's okay, Kevin Hart, we'll go step by step together.
- First, we do the **arithmetic**. Remember, parentheses first, then multiplication/division from left to right, and finally addition/subtraction.

```
5 * 7 >= 3 + 5 * (7 - 1) and not False
5 * 7 >= 3 + 5 * 6 and not False
35 >= 3 + 5 * 6 and not False
35 >= 3 + 30 and not False
35 >= 33 and not False
```

- Cool, now we finished arithmetic. Let's do **comparison** next. **(Mehran's voice) Dinggggg!** 35 is greater than 33: this is `True`.

```
35 >= 33 and not False
True and not False
```

- Next, we execute `not`. Remember, `not` flips the condition of something. Therefore, `not False` is `True`!

```
True and not False
True and True
```

- Lastly, we take care of `and/or`. `True` and `True` are…well, `True`.

```
True and True
True
```

- Therefore, we know this whole statement is True!

BOOLEAN IN VARIABLES

- We can also store **booleans** into **variables** (what?!)
- Think back to those magical variable suitcases from Lecture… remember how we could specify the *type* as stuff like `int`, `float`, and `string`? Recall that `bool` was also one of those!
- **Boolean Variables** are variables that store values which evaluate to either True or False.
- For example, if I wrote `x = 1 < 2`, we have:
  - Created a variable named `x`
  - Assigned (and stored) the value `1 < 2` to `x`
  - This evaluates to *True* since 1 is, indeed, less than 2. Therefore, the Boolean Value `True` is stored to `x`.
- What about `y = 5.0 == 4.0`? We have:
  - Created a variable named `y`
  - Assigned/stored the value `5.0 == 4.0` to `y`
  - This evaluates to *False* since 5.0 is not the same as 4.0. Therefore, the Boolean Value `False` is stored to `y`.
- You can also **directly set the value of a variable to a Boolean.**
- If I wrote `cs106a_is_cool = True`, I have:

  - Declared a variable named `cs106a_is_cool`
  - Assigned the value `True` to it
- What about `red_lightsabers_are_best = False`? I have:
  - Declared a variable named `red_lightsabers_are_best`
  - Assigned the value `False` to it

- How about another example? Let's say you designed a cool video game, and you want to prompt a user to choose if they want to play again (the answer is yes, of course!)
- Your code might look something like this:

```
play_again = input('Play again? "y" or "n"') == 'y'
if play_again:
  ...
```

  - We declare a variable named `play_again`.
  - We assign the value of an input to the variable.
    - Notice how we enclose the input with single quotation marks! This is because we have double quotation marks in our string. We want the program to recognize single quotation marks as the string, and not the double quotation marks.
    - To that input, we set it as "the same as" (==) 'y'. That means that if the user inputs `y`, the expression will evaluate to `True`.
  - Lastly, we type `if play_again:`. This statement may seem strange, but it simply means, "If `play_again` is True, then execute the body of the `if` statement."

Notes by Regina Gerbeaux

`for` Loops

INTRO TO FOR LOOPS IN PURE PYTHON

- We are already familiar with `for` loops in Karel, and we'll find they work very similarly in pure Python as well! (As we know, Karel is still "real" Python…that's why it works so similarly!)
- Here's an example of a `for` loop in Python!

```
public void run():
  for i in range(100):
    print("Python rocks socks!")
```

Our `for` loop reads, "Repeat the code in this loop 100 times."

- But what's really happening under the surface of a `for` loop? Let's take a magnifying lens to it.
- This is *really* what's happening in the code above:

```
i = 0
while i < 100:
  print("Python rocks socks!")
  i += 1
```

- *WAIT WHAT, A* `while` *LOOP?* You betcha. :)
- We start with the variable `i` (that counts the number of times the loop should run) and assign it a value of `0`. (We are **zero-indexing** it, which means Loop 1 starts at `0`. More on that in another lecture.)
- `while` `i`'s value is less than `100`:
  - The statement, "`Python rocks socks!`" will print
  - `i`'s value will update by adding `1` to itself.
- **Let's take a look at what the code does the first time.**

`i` has the value of `0`.

```
i = 0
while i < 100:
  print("Python rocks socks!")
  i += 1
```

Is `i` less than `100`? Yep, that is True!

Since it's true, let's `print` the string.

Lastly, let's update the value of `i`. `i` now has the value of `2`.

- And then we repeat that until `i` is no longer less than `100`!
- When `i` = 100, `i` is no longer less than `100`. Therefore, the loop will end.
- This means the range `100` represents values from `0` to `99`. Remember we set `i`'s value to `0` in the beginning. The loop running from `0` to `99` inclusive means it will run `100` times.

Problem Solving: Printing Even Numbers from 0 to 100 (Own Example)

Note: This example is **different from the example given in lecture. I decomposed this problem thinking Chris's question was to print *only* the even numbers from 0 to 100. If you're looking for the example from lecture, please scroll down to the next Problem Solving question.)

- Our objective is to print all the *even* numbers from 0 to 100. How would we do that?
- You know what the first step is:

  - We can use a `for` loop to go through numbers all the way up to `101` (`for i in range(101):`). Use `101` so it prints `100`.
  - Within that loop, we know that even numbers divide evenly into 2…so we can use the remainder operator! (`%`)
  - `if` a number dividing into 2 leaves a remainder of 0, we can print that number.
- Cool, here's the code!

```
def main():
  for i in range(101):
    if i%2 == 0:
      print i
# output: 0, 2, 4, 6...100
```

Problem Solving: Printing the First 100 Even Numbers (Lecture Example)

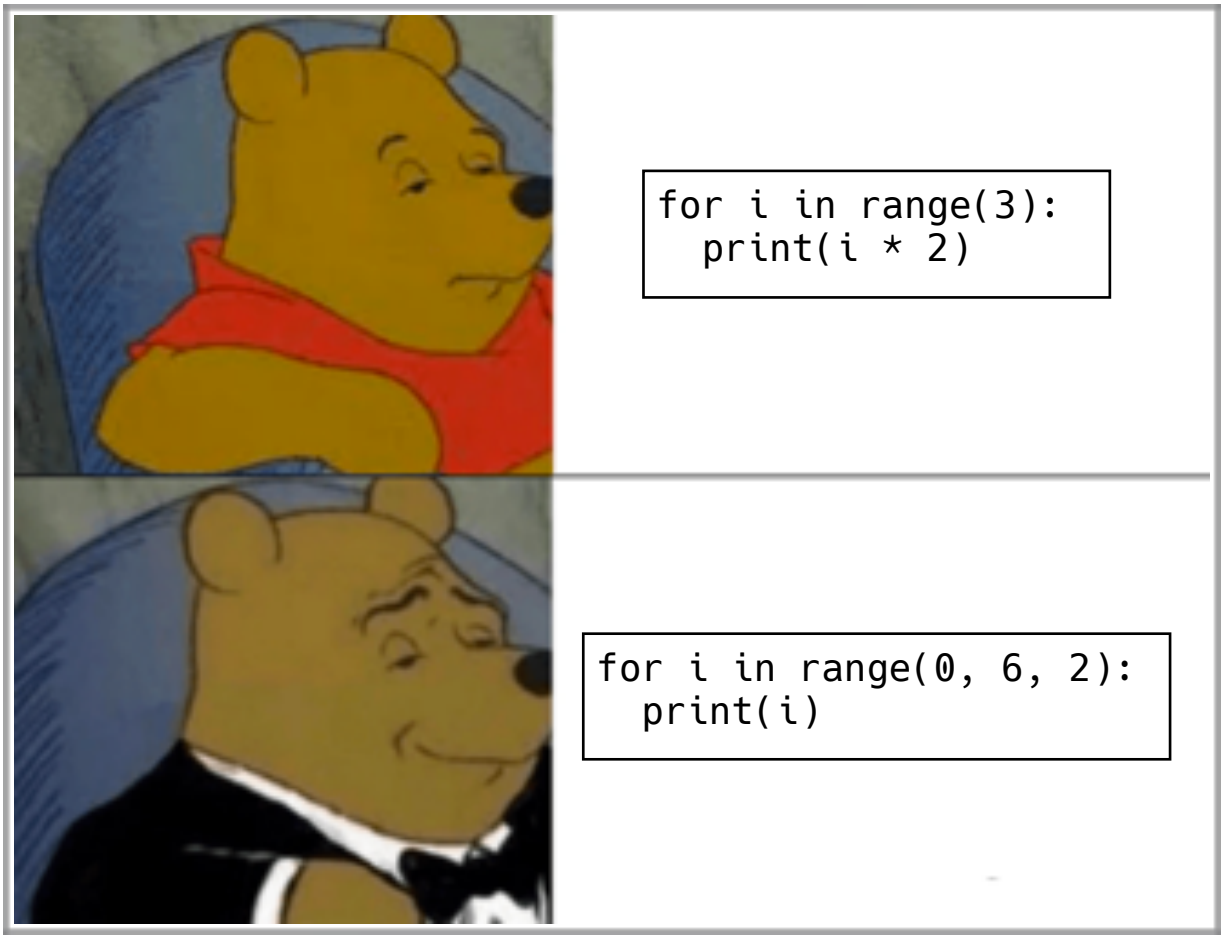
- Our objective is to *print the first 100 even numbers.* Decompose!
  - We can use a `for` loop to run our code `100` times.
  - Within that loop, we know numbers multiplied by 2 are even. So we can use the multiplication operator (`*`) and `print` the result!
- Here's our code:

```
def main():
  for i in range(100):
    print(i * 2)
# output: 0, 2, 4, 6...198
```

- Tada!

Alternate Solution: Print 3 Even Numbers

- Here's another cool way to print the first three even numbers.



- The **first** number, `0`, says we start `i` at `0`.
- The **second** number, `6`, says we stop before `6`.
- The **third** number, `2`, says we skip `i` by `2` each time.
- This means the first time we run the loop, `i = 0`, and we print `0`.
- The second time, we skip `i` by `2`, so `i = 2` and we print `2`.
- The third time, we skip `i` by `2`, so `i = 4` and we print `4`.
- Then, we skip `i` by `2` so `i = 6` now, but our stopping condition is to stop before `6`. Therefore, the loop ends.

Problem Solving: Game Show

- This is the problem we're going to solve on Monday (4/27/2020): `gameshow.py`!
- The program reads, "`Welcome to the CS106A game show! Choose a door and win a prize`"
- Then, you choose a door.
- The program tells you, "`You chose door *your-door-number-goes-here*`"
- And finally, the program tells you, "`You win $*prize money*`"

Homework for the weekend: Think about how this code would work, and decide which door you would choose for the best prize!

```
# logic for choosing a door

door = int(input("Door: "))
# while input is invalid
while door < 1 or door > 3:
  # tell user input is invalid
  print("Invalid door!")
  # ask for new input
  door = int(input("Door: "))
```

```
# logic for door prize decision

prize = 4

if door == 1:
  prize = 2 + 9 // 10 * 100

elif door == 2:
  locked = prize % 2 != 0
  if not locked:
    prize += 6

elif door = 3:
  for i in range(door):
    prize += i
```

CS106A Notes - Python

Week 3 Notes: Functions Revisited (4/27/2020 Lecture)

Recap on 4/24/2020 Lecture

BOOLEAN

- **Booleans** are the most simple **variable** type because they can only take on *True* or *False.*
- Here is an example of a **Boolean Variable**:

```
karel_is_awesome = True
```

- Above, we see the value `True` assigned to the variable `karel_is_awesome`.
- How about another example?

```
my_bool = 1 < 2
```

translates to:

```
my_bool = True
```

- The value of `my_bool` evaluates to `True` because `1 < 2` is true. The `<` is a **boolean expression**.
- How about **Boolean Operations**?
  - As a reminder, boolean operations include `and`, `or`, and `not`.
  - If `a = True` and `b = False`, consider the following examples:

```
both_true = a and b

either_true = a or b

opposite = not a
```

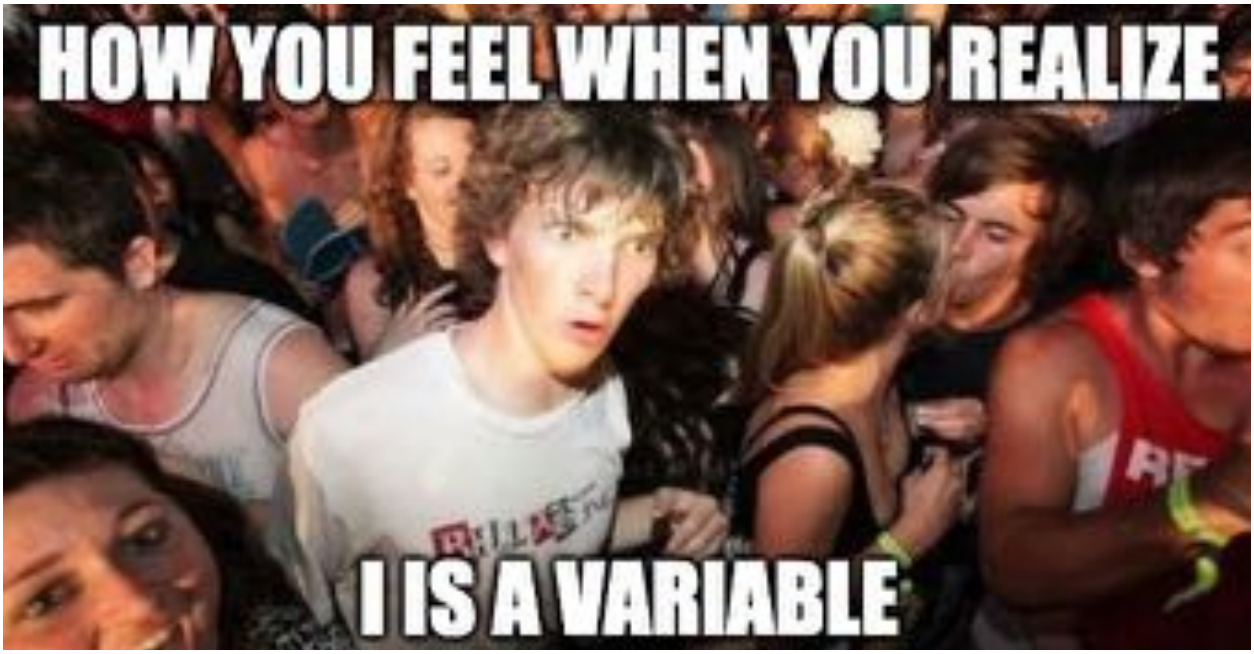
Evaluates to `False` since **both** have to be true for it to be true.

Evaluates to `True` since **either** can be true for it to be true.

Evaluates to `False` since it's the opposite of `a`. `a = True` so `not a = False`.

FOR LOOPS

- The `i` in a `for` loop is a variable that represents an **index**. It keeps track of how many times the loop will run.
- Because `i` is a variable, you can also use it in the body of your text.



- For example:

```
for i in range(10):
  print(i)
```

This prints the value of `i` while the `for` loop runs 10 times.

```
0
1
2
3
4
5
6
7
8
9
```

This is the output.

Notes by Regina Gerbeaux

Problem Solving: Game Show

GAME SHOW STEP BY STEP

Here is the code to the CS106A Game Show problem!

```
def main():
  print("Welcome to the CS106A Game Show!")
  print("Choose a door and pick a prize")
  print("--------------------")

  # PART 1: Get the door number from user
  door = int(input("Door: "))
  while door < 1 or door > 3:
    print("Invalid door!")
    door = int(input("Door: "))

  # PART 2: Compute the prize
  prize = 4
  if door == 1:
    prize = 2 + 9 // 10 * 100
  elif door == 2:
    locked = prize % 2 != 0
    if not locked:
      prize += 6
  elif door == 3:
    for i in range(door):
      prize += i

  print("You win: " + str(prize) + ' treats')
```

Ask for `door` #

Tell user input is invalid and prompt for a new input

`prize` starts out with a value of 4.

See below for code explained.

See below for code explained.

If user chooses Door 3, prize adds value of `door` (3) to `prize` (4), so `prize` = 7.

This prints to the Console how many treats our doggo has won.

Here is what happens at Door 1:

```
prize = 2 + 9 // 10 * 100
prize = 2 + 0 * 100
prize = 2 + 0
prize = 2
```

Next, do *.

`//` (int division) chops off any #s after the decimal. Since 9/10 is 0.9, `9 // 10` chops off the decimal point so `= 0`.

And here's what happens at Door 2:

```
locked = prize % 2 != 0
if not locked:
  prize += 6
```

`locked` var declared. Value is `prize % 2 != 0`, which evaluates to a **Boolean**. `prize = 4`, and `4 % 2 = 0`, Therefore, `0 != 0` is `False`.

`if not locked` asks `if not False`, so `not False` becomes `True`. Therefore, we add 6 to the value of `prize`, and `prize` is now `10`.

Notes by Regina Gerbeaux

Intro to Functions in Python

FUNCTIONS TO DEFINE NEW COMMANDS

- One of the first things we learned in Karel was how to define new commands (remember `turn_right()` was `turn_left()` three times?)
- Just like we defined commands in Karel, we can also define new commands through functions in Python. Think of `turn_right()` as a **function**. The cool thing about functions in *pure* Python, however, are that we can write functions that **accept input from a user**, **give an output back to the user**, and we can **trace functions** using stacks (stepping thru program step by step like a computer does)

CALLING FUNCTIONS

- In Karel, when we were **typing commands** like `move()`, we were actually *calling the function* to run in our console. We also **created functions** like `turn_right()` by defining how they work.
- Similarly, in pure Python, we have already been *calling functions*:

```
input("Gimme ur number: ")
print("Fuck off")
math.sqrt(25)
float("0.42")
```

All of these are functions. They all take information (*input*), and return something (*output*).

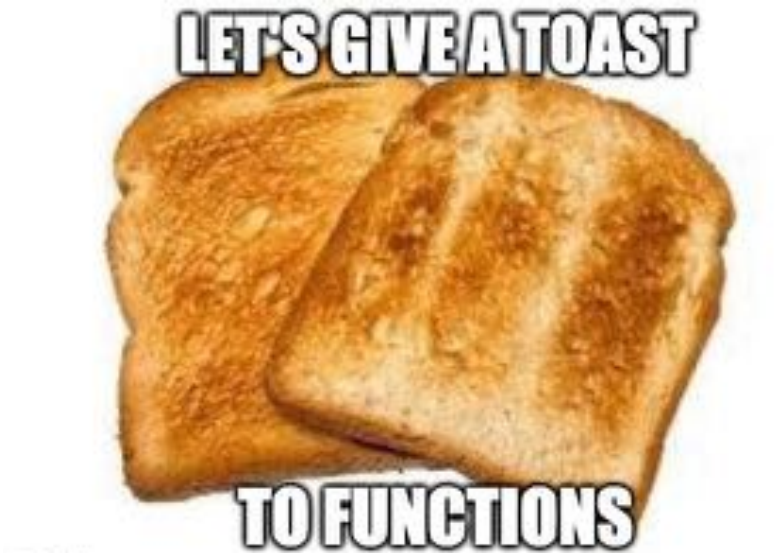
- **The main difference with functions between Karel and Python is that Python can take in data and also return data.**

FUNCTIONS (AS REPRESENTED BY A TOASTER)

But why use a toaster when you could just use a lightsaber? (Or something. Whatever.)

Toaster	**Function**
A ***toaster*** does something.	A ***function*** does something.
You put ***something*** into a toaster.	You put ***inputs*** into a function.
That "something" you put into a toaster is ***something you want to put in*** (like, don't put mashed potatoes in there. Unless you want to. But that's kind of weird.)	A ***parameter*** is the thing you specify on ***what you are going to put in*** to the function.
The toaster ***does the thing you want*** to the thing you put in, and gives you back the thing you want. (e.g.: bread becomes toast)	A function ***does the thing you want*** to the input you gave, and gives you back an ***output***. That output is called a ***return***.
The toaster ***should be able to work for other stuff you want to put in*** (not like mashed potatoes, but like bagels!) - your toaster would be pretty poop if it didn't work for bagels (unless bagels aren't your thing.)	A function ***should be able to work for other stuff you want to put in*** — it should work so that you give the same function different inputs and it gives back different outputs.

- **You don't necessarily need to know *how* a function works, but you *do* need to know *what* the function does.** (Just like you don't necessarily need to know *how* a toaster works, but you *do* need to know *what* the toaster does.)



Notes by Regina Gerbeaux

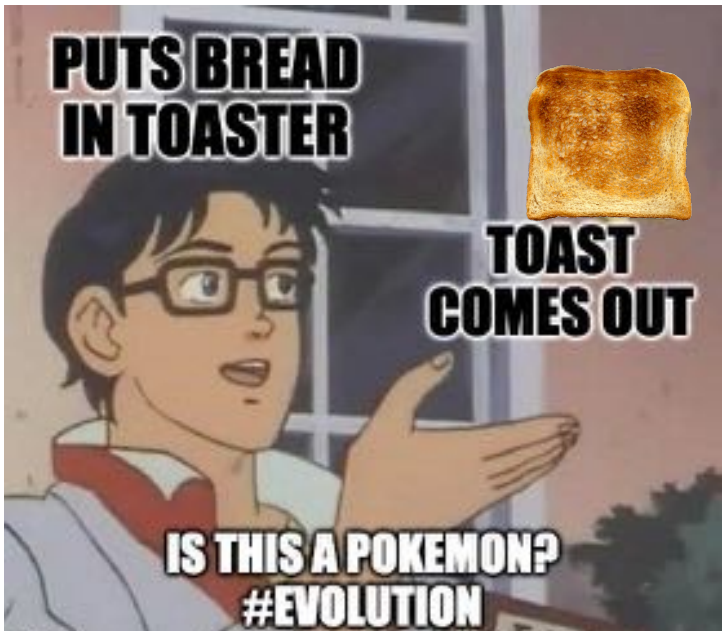
Anatomy of a Function

STRUCTURE OF A FUNCTION

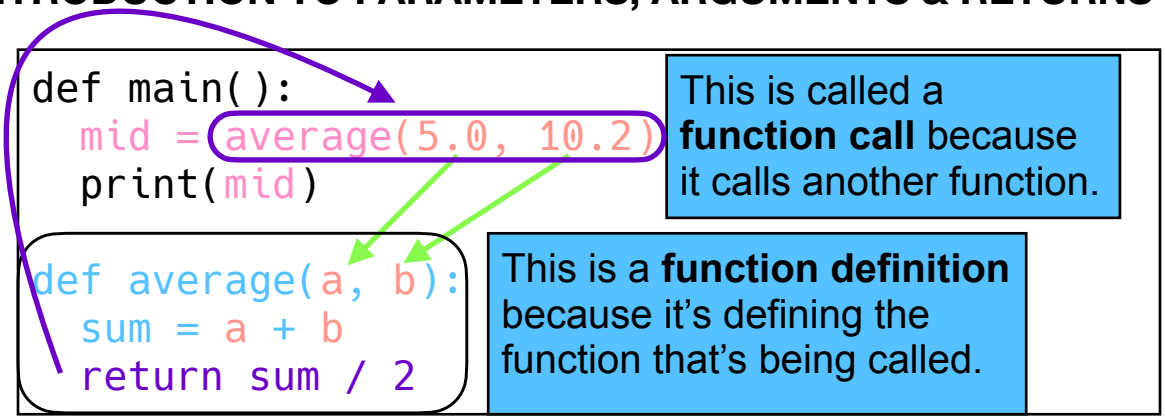
```
# python code

def name_of_function (param):
  body of statements

  # optional
  return value
```

```
# karel code

def name_of_function( ):
  body of statements
```

- As you can see, functions in Python and functions in Karel are very similar (this is because Karel is written in Python!)
- The differences are highlighted in pink. The Python function contains two new concepts: **parameters** and **return** statements.
  - Note: not all functions need an input, and not all functions need an output.
- **Parameters** are what you *give* to the function (in our toast analogy, this is represented by bread.)
- **Return** statements are what you *get back* (what is "returned" harhar) from the function (in our toast analogy, this is represented by toast.)
- To recap, **name** is what we call the function, **parameters** is the info passed into function, and **return** is the info given back from function.



INTRODUCTION TO PARAMETERS, ARGUMENTS & RETURNS



Main Function

- In `main()`, we see `average()` called in the variable `mid`.
  - A variable named `mid` is declared. `mid`'s value is **whatever `average()` returns**.
  - We see `5.0` and `10.2`: these are the **arguments** passed into `average`. These are the **inputs given**.

Average Function

Then, we have the `average` function (written in **blue**.)

- `average` is the **name** of the function.
- `average` takes in `a` and `b` as the **parameter** (**inputs expected**.)
- In `average`'s **body**, a variable named `sum` is declared. `sum` adds the two values given *from* the parameter.
- Finally, we have a `return` that gives back the value of `sum / 2`.

Calculating Average with Arguments from Mid

- Recall that `mid` calls the `average` function with two values given: `5.0` and `10.2`. These values from `mid` are passed into `average` as `a` and `b` respectively (so `5.0` is `a`, and `10.2` is `b`.)
  *Note, you can see how it's passed marked by green arrows above.

- `average` can now execute its function with the inputs given, and **return** a value.
  - We know that `5.0 + 10.2 = 15.2`, so the value of `sum` is `15.2`.
- Lastly, we have a **return** statement: `sum / 2` will be given to whoever calls it.
  - In this case, `15.2 / 2 = 7.6`, so `7.6` will be given back to `mid` in `main()`.

Returned Statement

- `7.6` is **returned** to `mid`, and that is now the *value* of `mid`.
- Therefore, when we `print(mid)` in `main()`, `7.6` will print.

A Few Notes:

- `main()` is also a function, it just doesn't have parameters or returns. Recall that functions don't always have to have parameters or returns because they don't always give arguments or take returns.
- **Parameters allow functions to accept input.** They let you provide a function with some information when you're calling it.
- **Returns are not the same thing as printing**. Returns *give* a value back to whatever calls it. This means they're not necessarily printed.

More Functions, More Examples

NO PARAMETER, NO RETURN EXAMPLE

```
def print_intro():
  print("Welcome to class")
  print("It's the best part of my day.")

def main():
  print_intro()
```

Print Intro Function

We have the `print_intro()` function (written in **blue**.)

- `print_intro` is the **name** of the function.
- `print_intro()` **takes no parameters** (in other words, it does not receive any input from anywhere else.)
- In `print_intro`'s **body**, there is some text printed.
- Notice **there is no return statement**.

Function Caller

- In `main()`'s body, there is one statement — wow, `print_intro()` is called!
- That means when `main()` runs, `print_intro()` is called.

Execution

- We run `main()`, and `print_intro()` is called.
- `print_intro()` runs and prints, "Welcome to class" and "It's the best part of my day." into the Terminal. The end!

Note: *this is super similar to Karel when we define the function* `turn_right()` *separately and then call it within* `main()`. *It's similar because recall Karel does not have parameters and return statements! If you're struggling, write out some Karel code and try to explain it the way Python functions are explained above.*

PARAMETER EXAMPLE

```
def print_opinion(num):
  if (num == 5):
    print("I love 5!")
  else:
    print("Whatever")

def main():
  print_opinion(5)
```

Print Opinion Function Definition

We have the `print_opinion()` function.

- `print_opinion` is the **name** of the function.
- `print_opinion` takes in `num` as the **parameter** (in other words, it expects an *input* of `num`.)
- In `print_opinion`'s **body**, we see an if-else statement.
  - If the value of `num` is equal to `5`, "`I love 5!`" will print.
  - Else, "`Whatever`" will print. (Rude!)

Function Caller

- In `main()`, we see `print_opinion()` called in the body.
- Notice the **argument** (input given) `5` to `print_opinion()`. This is passed to the parameter (input expected) in `print_opinion`.

Execution

- We run `main()`, and `print_opinion()` is called.
- The argument `5` is passed into the parameter `num` in `print_opinion()`, so in this instance, `num = 5`.
- Since `num = 5`, the statement, "`I love 5!`" will print. The end!

PARAMETER AND RETURN EXAMPLE

```
def meters_to_cm(meters):
  return 100 * meters

def main():
  result = meters_to_cm(5.2)
  print(result)
```

Meters to cm Function Definition

We have the `meters_to_cm()` function.

- `meters_to_cm` takes in `meters` as the **parameter** (in other words, it expects an *input* of `meters`.)
- In `meters_to_cm`'s **body**, we see a **return** statement. It will *give back* the value of whatever `100 * meters` is *to the place that called* `meters_to_cm` *in the first place*.

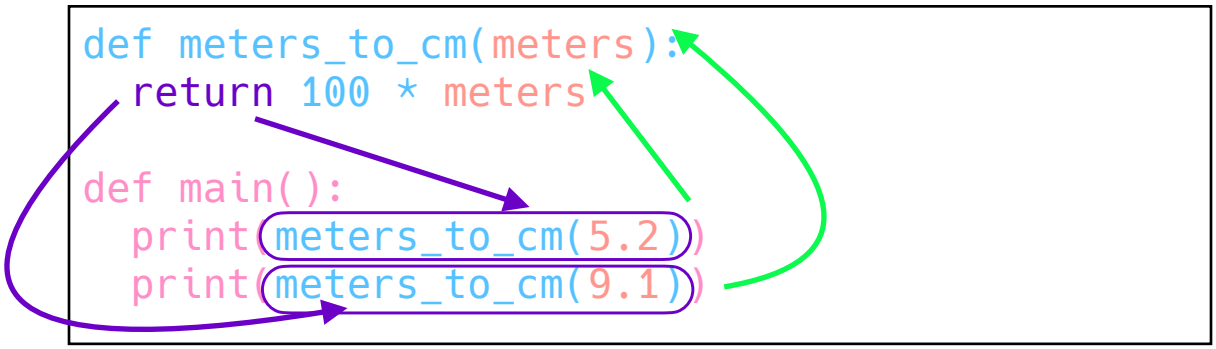
Function Caller

- In `main()`, the variable `result` has a value that is the result of `meters_to_cm()` with the **argument** of `5.2`.
- `5.2` is passed to the parameter (input expected) in `meters_to_cm`.

Execution

- We run `main()`, and `meters_to_cm()` is called in the variable `result`.
- The argument `5.2` is passed into the parameter `meters` in `meters_to_cm()`, so in this instance, `meters = 5.2.0`.
- Since `100 * 5.2 = 520.0`, `520.0` is returned as the value of `result`. This does *not* print: it just gives the value back to `result`!
- When we `print(result)`, `520.0` will print. The end!

PARAMETER AND RETURN EXAMPLE EXTENDED



**Note that this is the exact same function definition as above, except now we have a different main function call.*

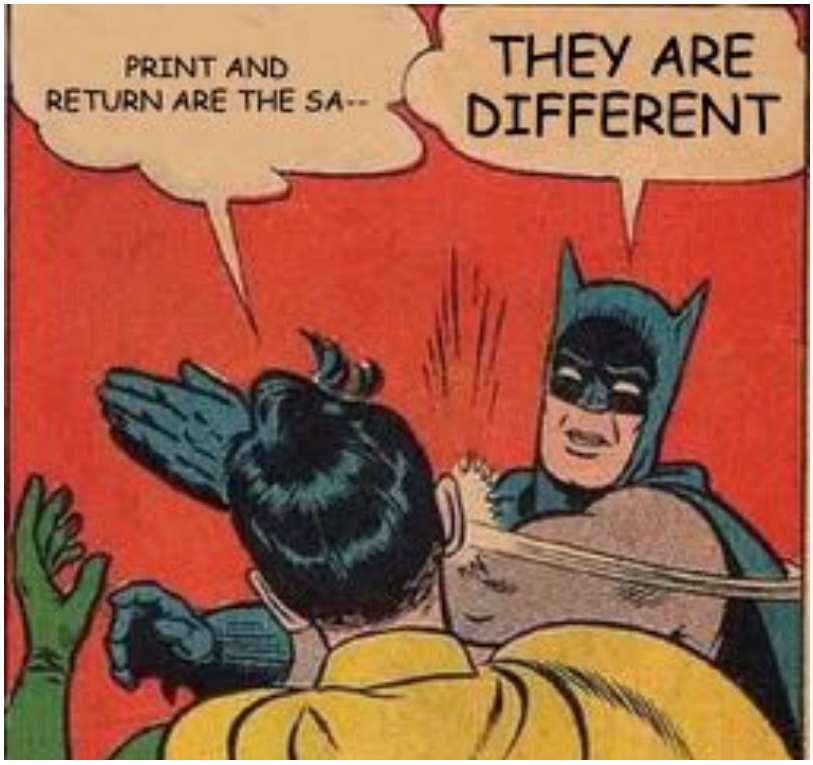
Main Function Call

- In `main()`, we have two `print` statements. Both are calling `meters_to_cm()` with the **argument** of `5.2` and `9.1` respectively.
- First, `5.2` is passed to the parameter in `meters_to_cm`. The function runs, and returns the value of `100 * 5.2`, which is `520.0`.
  - Therefore, `520.0` will `print` in the Terminal.
- Then, `9.1` is passed to the parameter in `meters_to_cm`. The function runs, and returns the value of `100 * 9.1`, which is `910.0`.
  - Therefore, `910.0` will `print` in the Terminal.

Execution

- We run `main()`, and `meters_to_cm()` is called in the body.
- In the first `print` statement, `5.2` = `meters` in `meters_to_cm()`, so in this instance, `520` is returned to the value of `result`. Then, we print the value of `result` and `520` appears in the Terminal.
- In the second `print` statement, `9.1` = `meters` in `meters_to_cm()`, so in this instance, `910.0` is returned to the value of `result`. Then, we print the value of `result` and `910.0` appears in the Terminal.

DIFFERENCES BETWEEN RETURN AND PRINT



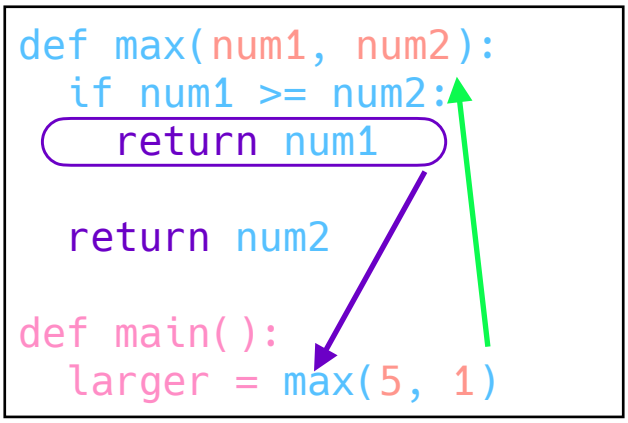
Let's elaborate a little more on the difference between `print` *and* `return`. What is the difference between the following two examples?

```
def case1(meters):
  return 100 * meters
```

```
def case2(meters):
  print(100 * meters)
```

- Besides the names being different (harharhar), one contains a **return** statement, and the other contains a **print** function. Let's go over what they each do.
- In `case1`, `return` gives the function's result back to the caller. The caller can do whatever it want to that result (including printing it), but returns do NOT automatically print!
- Once the function returns a value, it *forgets* the values generated. (Therefore, you can call the function again and again with different arguments, and each time it would return different values.)
- In `case1`, `print` will actually print the result onto the Terminal.

MULTIPLE RETURN STATEMENTS



Max Function Definition

- We have the `max()` function. `max` has `num1` & `num2` as parameters.
- In `max`'s body, there's an if-else statement. If `num1` is greater than or equal to `num2`, `max` will return `num1`. Else, it will return `num2`.

Main Function Call

- In `main()`, the variable `larger` has a value that calls `max()`. It provides the **arguments** `5` and `1`, which get passed into `max` as `num1` and `num2` respectively.
- Once `max()` returns a value to `larger`, `larger` stores that return value as its own value.

Execution

- We run `main()`, and `max()` is called in the variable `larger`.
- The arguments `5` and `1` are passed into the parameters `num1` and `num2` respectively (so `num1 = 5` and `num2 = 1`.)
- `max` runs and checks if `5 >= 1`. Since this is True, `max` returns the value of `num1` (which is `5`) to `larger`.
- `larger` now has the value of `5`.

Problem Solving: IO

The objective of IO is to print a really beautiful piece of art formed by the characters “i” and “o”. See the video lecture for what it looks like.

```
import math

MAX_SPACES = 20

def main():
     for i in range(MAX_SPACES):
          print_io_line(i) #increases num of spaces
     for i in range(MAX_SPACES):
          print_io_line(MAX_SPACES - i) #decreases num of spaces

# this function prints a single line of the io program
# first, print space_before_io number of spaces
# then, print “io”
def print_io_line(space_before_io):
  print_n_spaces(space_before_io)
  print(‘io’)

def print_n_spaces(n):
  for i in range(n):
    print_no_new_line(‘ ‘)# prints n spaces with no new line


# this function takes in string and prints everything on the
same line.
def print_no_new_line(to_print):
  print(to_print, end=“”)
```

CS106A Notes - Python
Week 3 Notes: Functions Revisited (4/29/2020 Lecture)

Recap on 4/27/2020 Lecture

FUNCTIONS REVISITED

- This is a **function definition**. It defines a function.

```
def average(a, b):
    sum = a + b
    return sum / 2
```

name: `average` | parameters: `a, b` | function body: `sum = a + b` | return statement: `return sum / 2`

if you need review on any of these definitions, refer to 4/25's notes.

- Then here, we see a **function call**. It calls another function. Notice here that we're calling the function we defined above.

```
def main():
    avg = average(5.0, 10.2)
    print(avg)
```

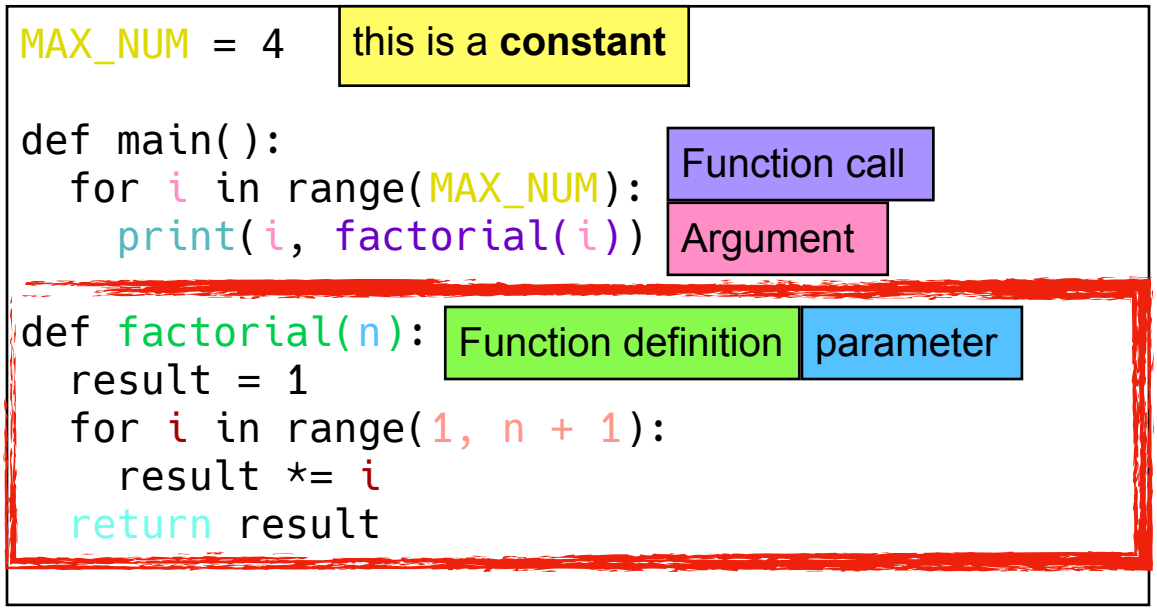
variable: `avg` | argument: `5.0, 10.2` | function call: `average(5.0, 10.2)` | prints result of average: `print(avg)`

- Focus: **Parameters** let you provide a function with information (input) when you call it. When you **call** the function, the information you pass in is an **argument**. *(For clearer examples, refer to 4/25's notes: the green arrows in those notes illustrate how arguments get passed into parameters.)*

Factorials

INTRODUCTION TO FACTORIALS

- Here is a full program as given by Mehran. This is about **factorials**. The objective of this program is to count all the way up to `n` and multiply the numbers. In other words, `n` factorial, also written as `n!`, can be defined as `1 * 2 * 3 * ...` until we reach `n`.
  - For example, if `n == 4`, `4! = 1 * 2 * 3 * 4`, so `4 factorial` has a value of `24`.



Within MAIN:

- `MAX_NUM` is a **constant** with the value of `4`. Any function can refer to this value because it has a **global scope**.
- In our `main()` definition, we have a `for` loop.
  - Recall that `i` is a **variable** that we have access to.
  - Notice the `for` loop range is set to the constant we declared.
- Each time the `for` loop runs, it will **print** the current value of `i`

- In addition, notice we will also print the value of `factorial( i )`. This is a **function call** that will call the `factorial` function. It passes in the current value of `i` as its **argument**.
- **Note: the reason why the print statement has two values with a comma is because it can print both statements on the same line with a space in between.**

Within FACTORIAL:

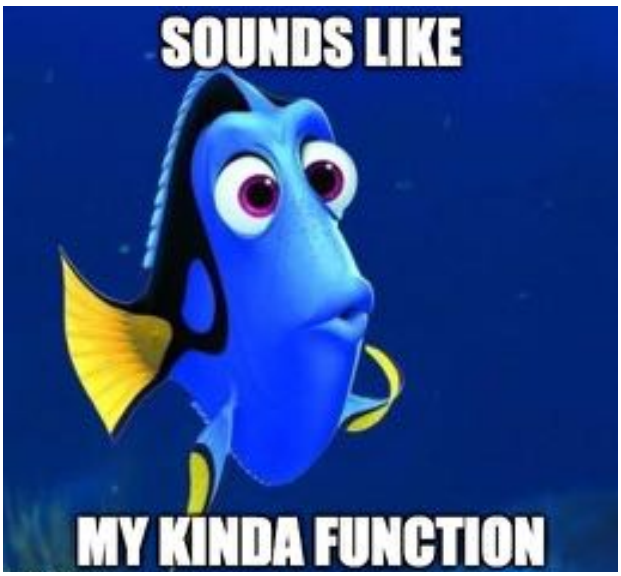
- Now, we can move onto the **function definition**, `factorial(n)`.
  - `factorial` has a **parameter** set to the value `n`.
  - Recall in our **function call**, we've passed the value of `i` as the **argument**. Therefore, `n` in `factorial( )` has the value of `i`.
- **This function has no recollection it has ever been called, so when you run your `for` loop in `main( )` and it calls this function, no variables from the previous loop are saved.**
- We declare a variable called `result` and assign it a value of `1`.
- Then, we have a `for` loop that set the **range** (value of `i`) beginning at `1` and counting all the way up to `n + 1` (but not inclusive.)
  - In the body of the loop, the value of `result` will multiply by `i`.
  - ***This instance of `i` is different from the instance of `i` in main. They have different scopes.*** See "vocabulary" for more info.

- Once that loop is done, the result is **returned** to the function caller, which will then be printed in the **print** statement:

```
0   1
1   1
2   2
3   6
```

VOCABULARY

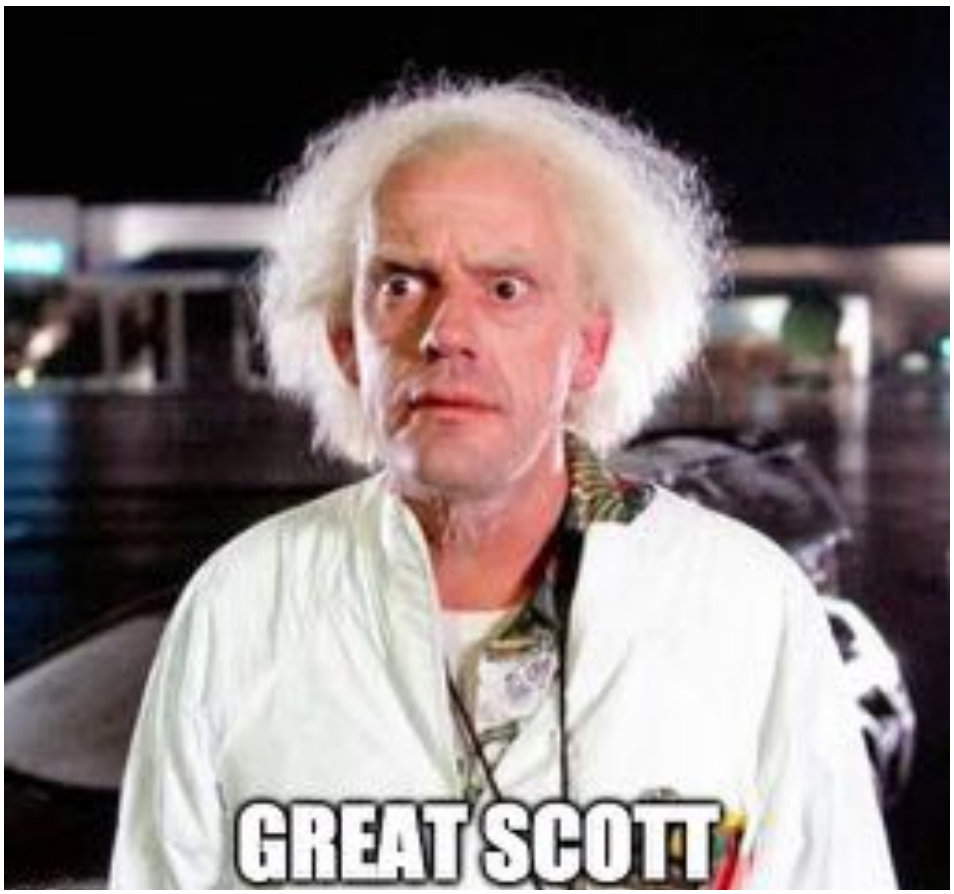
- **Scope** refers to the space where variables exist, and that informs us which variables the program can access in specific places.
  - As we remember, **constants** are pieces of information that live outside of any function, and therefore *all* functions have access to it. This is referred to as the **global** scope.
  - What about stuff on a smaller scope? Remember in the example above, we have the variable `i` declared two separate times in two different functions (once in `main()`, and again in `factorial()`.) They are two separate variables that have the same name but they "live" in different places. These are **local variables**.
- **Stack frame** is another way to refer to how a function is like a *frame* that contains *local variables* inside it. If you look at the code from the previous example, I've framed the stack frame of `factorial()` inside a red frame :)
- **No Old Values:** Each time you call a function, new memory is created for that call. You pass in parameter values (known as **arguments**) to the function. All **local variables** start fresh.



DocTests

INTRODUCTION TO DOCTESTS

- A **DocTest** is a way in Python to be able to test functions one at a time while you're writing code.
- In the factorial function we've traced through, we can set up Doctests to run our program with test cases to make sure our code is working the way we want them to.



- To write a doctest, we use the following format:

```
"""
>>> name-of-function-you'll-test(argument)
expected output
"""
```

- If we wanted to write three doctests for `factorial()` from the previous example, this is what it could look like:

```
def factorial(n):
    """
    This function returns the factorial of n
    Input: n (number to compute factorial of)
    Returns: value of n factorial
    Doctests:
    >>> factorial(3)
    6
    >>> factorial(1)
    1
    >>> factorial(0)
    1
    """
```

comment quotations

>>> denotes a Doctest | name of function | argument

expected output

comment quotations

- Doctests are good to test *edge cases* (tricky scenarios) to make sure your code is general enough to work even for them.

RUNNING DOCTESTS

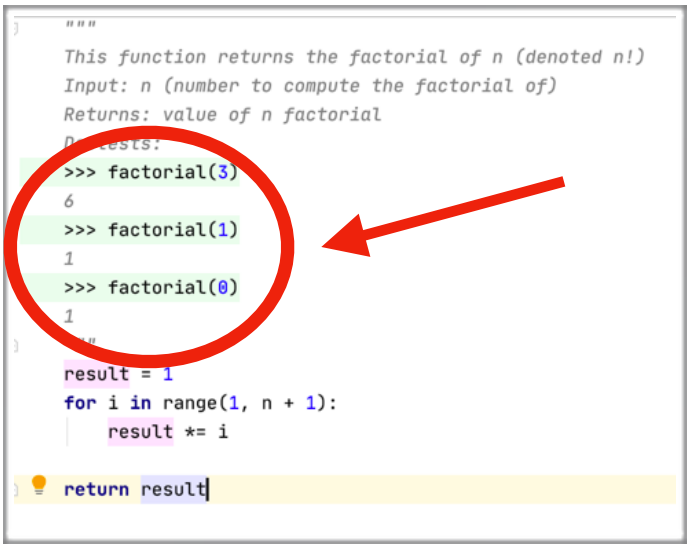
- To run the Doctests you've created, run this in your Terminal:

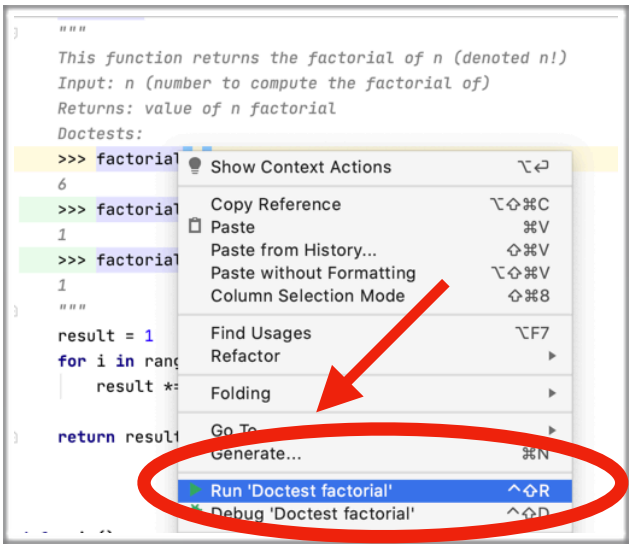
```
python3 -m doctest name_of_file.py
```

stands for module

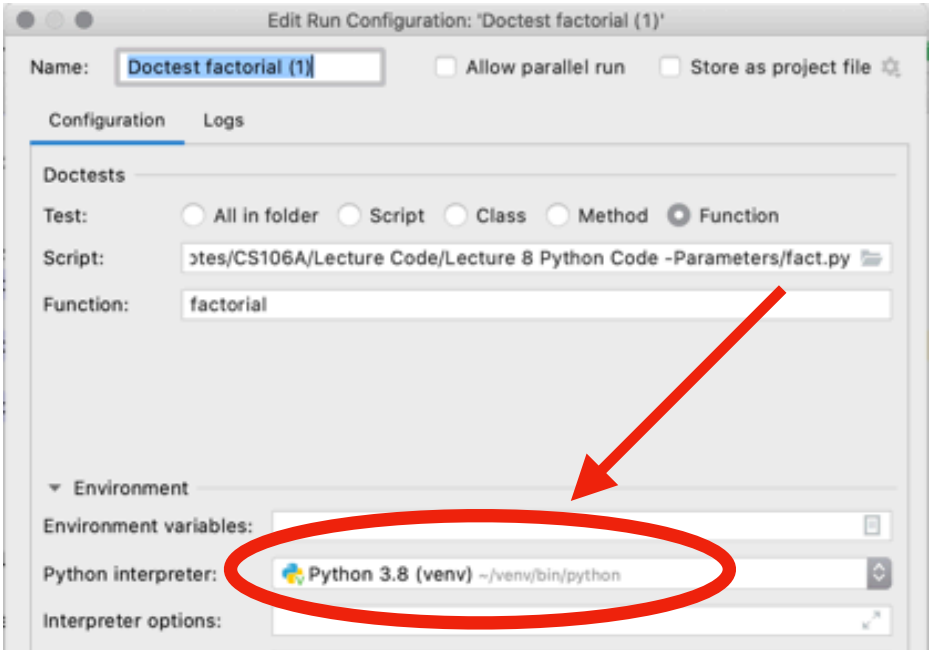
specify name of file

- Once you run this, it will print <u>any failures your Doctests have</u> if there are any. Otherwise, *nothing* will print if there are no errors.
- If you want to see all of the info that's being run, you can append the above statement with `-v` which stands for **verbose**, and it will show you all of the tests it runs.
- Additionally, in PyCharm, if you **right click the highlighted green text where your doctests are, you can run your tests there.**





- Make sure your Doctests are configured to run with a **Python3** Python interpreter.



- Doctests are also great for **regression testing** (if you change something and you break your code, when you run your doctests again you can see which answers changed.)
  - This can be especially helpful for debugging your code and figuring out where you have errors in your program.

Passing Primitives

TRACING A PROGRAM

- To **trace** a program means we'll go through it step by step to find our errors. This is good practice for looking through buggy code!

…Sorry dude, not that kind of buggy.

- In this following example (of a bad function), we'll trace through the program together.

```
def main():
  x = 3
  add_five(x)
  print("x = " + str(x))

def add_five(x):
  x += 5
```

yikes.

- Let's go through the **stack frame** of `main()`:
  - The variable `x` has a value of `3`.
  - And then we call the function `add_five()` and pass a parameter of `x`! Recall that when we pass an argument to a function's parameter, it is a **copy** of that argument.

  - In this case, we are passing a **copy** of `x`'s value to `add_five()`, so we pass a copy of the value `3`.

- Now, let's go through the stack frame of `add_five()`:
  - We have the parameter `x`, and passed the value `3` from `main()` as an argument.
  - `add_five()` computes and adds 5 to `x`, so `x = 8` now.
  - However, notice **we have no return statement.** Since `add_five()` is now done, we exit out of this stack frame *without passing the updated value*.
  - This is problematic since **called functions don't retain memory.** Once it finishes running, `add_five()`'s memory is wiped.

- Now, we go back to the stack frame of `main()`:
  - Since `x` from `add_five()` is a copy of `x` from `main()`, the `x` from `main()` still holds the value of `3`. It didn't update!
  - Therefore, `main()` will print `x = 3`.

- Yikes! That's not what we wanted. Let's make the function *not* buggy: This will return what we wanted: `x = 8`.

```
def main():
  x = 3
  x = add_five(x)
  print("x = " + str(x))

def add_five(x):
  x += 5
  return x
```

x is declared.

Update the value of x!

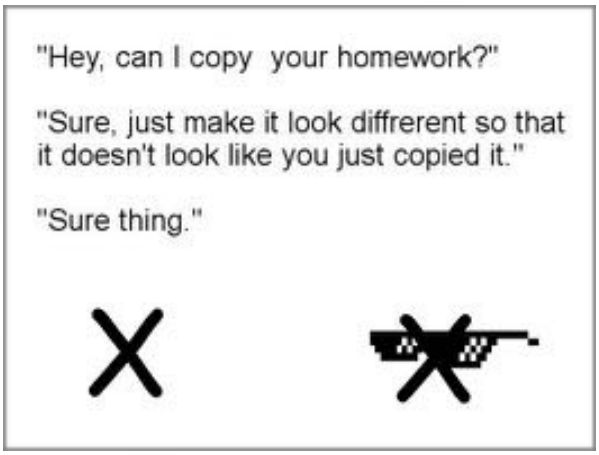
Prints with the updated value of x!

Return updated value of x!

OTHER STUFF TO REMEMBER

- For primitive types, ***copies* of values** are passed as arguments.

- Variables passed as parameters of the called function will not change when you modify it in the called function
- If you want to update the variable, the called function will return the updated value to the caller, and the caller can then update to the new value. This is called **assigning** over the variable (when you pass back some information from that called function and store it back into the original variable in the caller.)



- **Last Word of Warning: *Avoid writing functions within other functions.***
  - These are called nested functions which are technically legal, but often a sign at the start that *you are confusing function definition and function call.*
  - It's imperative to keep your functions separate for an easier time debugging and following your logic.

```
def main():
  print("hello world")
  def say_goodbye():
    print("ok bye now")
```

Here's our function within another function. Instead of this….

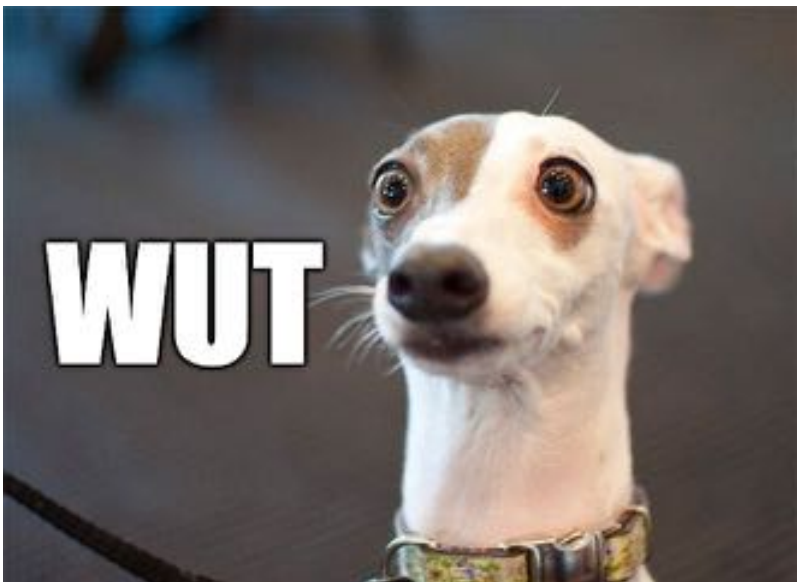
```
def main():
  print("hello world")
  say_goodbye()

def say_goodbye():
  print("ok bye now")
```

Woot woot we defined the function here instead. Do this instead.

Problem Solving: `calendar.py`

GIMME THE WHOLE BURRITO (BURRITOS > ENCHILADAS)

- Here's an example of a Python program that uses **functions, return statements, Booleans, control flow, parameters, and doctests**.



- `calendar.py` prints out a calendar for the entire year. 😱
- It makes sure that all twelve months show up, all the numbers for dates show up in the right place, and accounts for *leap years*.

Step One: Declare our Constants.

```
NUM_MONTHS = 12
NUM_DAYS_IN_WEEK = 7
```

Months and days in a week are constant — they don't change.

Step Two: Check if a Year is a Leap Year or Not.

```
def is_leap_year(year):
    """
    Returns Boolean indicating if year is a leap year.

    Doctests:
    >>> is_leap_year(2001)
    False
    >>> is_leap_year(2020)
    True
    >>> is_leap_year(2000)
    True
    >>> is_leap_year(1900)
    False
    """
    return ((year % 4 == 0) and (year % 100 != 0)) or
        (year % 400 == 0)
```

This is a predicate function, which return *True* or *False*.

- We've defined some Doctests to make sure our code can tell if a year is indeed a leap year or not.
- Leap years are *either* divisible by 4 **and** not divisible by 100, **or** divisible by 400.
- To check divisibility, we can look for *remainders* left:
  - If a number is divisible by 4, it will leave 0 remainder, and
  - If a number is not divisible by 100, it will leave a remainder.
- To find remainders, recall we use the remainder/modulo `%` sign.
- Notice we use parentheses to make sure our operations are very clear. Parentheses get *precedence*, so our program will execute `(year % 4 == 0)` first, then `(year % 100 != 0)`, and finally `(year % 400 == 0)`.
- We also use the `and` and `or` logical operators to make sure our program checks either *both* conditions with the `and` statement are true, or the second statement is true.

Step Three: Check Number of Days in Month.

```
def days_in_month(month, year):
  """
  Returns the number of days in the given month and year.
  Assumes that month 1 is Jan, month 2 is Feb, and so on.

  Doctests:
  >>> days_in_month(4, 2020)
  30
  >>> days_in_month(2, 1900)
  28
  """
  # Days in February depends on if it's leap year or not
  if month == 2:
    if is_leap_year(year):
      return 29
    else:
      return 28
  # April, June, September, and November have 30 days
  elif month == 4 or month == 6 or month == 9 or month == 11:
    return 30
  # All other months have 31 days
  else:
    return 31
```

- Our function `days_in_month` has two parameters: `month` to check which month we're evaluating, and `year` for our February edge case (since February's # of days will depend on if it's a leap year or not.)
- Again, we've defined some doctests to make sure our function works. Notice we test (2, 1900) to make sure our code works for this special edge case.
- Going through our code:
  - Using an `if` statement, if the month is February (`month == 2`), there are either 28 days or 29 days. We check this with an inner

`if-else` statement to check if the year is a Leap Year or not. Lastly, we `return` the value.

- Next, we chain an `elif` statement to our `if` statement.
  - If it's not February (`month == 2`), *else if* it's April, June, September, or November (where `month == 4`, `month == 6`, `month == 9`, or `month == 11`), there are 30 days. Then, we `return` the value.
- Finally, the rest of the months (January, March, May, July, August, October, and December) have 31 days.
  - We can cover these months by simply using an `else` statement and `returning` the value.

Step Four: Print Month Header.

```
def print_month_header(month):
  # prints header for a given month in the calendar
  print("Month #" + str(month))
  print("Sun Mon Tue Wed Thu Fri Sat")
```

- This step prints a header for the given month in the calendar.
- `print_month_header` accepts a parameter of `month`
- It will print the phrase `Month #` concatenated with the stringified version of `month`. (e.g.: if `month == 2`, then `Month #2` will print)
- Finally, it prints the header with the dates

Step Five: Format Numbers.

```
def format_number(num):
  """
  Formats a one or two digit integer to fit in four spaces.
  Returns a string of the formatted number string.
  """
  result = " " + str(num) + " "
  if num < 10:
    result = result + " "
  return result
```

- We want our numbers to print neatly in our Terminal, so we'll have to add some extra space for the #s that are smaller than two digits.
- We have `num` a parameter (since we'll pass in the date as a number)
- We create a variable named `result` that concatenates one space on each side of the stringified version of `num`.
- Then, we add an `if` statement to check if `num` is less than 10 (because that covers all single-digit numbers.) If it is, we'll update the value of `result` to add an extra space to fix the formatting.
- Lastly, we'll return the `result` (to send back to the caller function)

Step Six: Print the Calendar for a Given Month and Year.

This function will:

1. **Print a daily calendar for a given month and year,**
2. **Determine the day the month begins on, and**
3. **Return the day for the 1st of the *following* month.**

```
def print_month(first_day, month, year):
  """
  Prints daily calendar for given month and year. Also determines
  which day month begins on (day the 1st of the month falls on)
  (0 = Sunday, 1 = Monday, ..., 6 = Saturday)

  Returns day of week that the FOLLOWING month starts on
  """
  print_month_header(month)
  days = days_in_month(month, year)

  # Print leading space before first day in this month
  for i in range(first_day):
    print("    ", end="") #prints four spaces per day

  # Print numbers for all the days in the month
  for i in range(0, days):
    print(format_number(i + 1), end="")
    # Add a new line at end of the week
    if ((first_day + i) % NUM_DAYS_IN_WEEK) == (NUM_DAYS_IN_WEEK - 1):
      print("")
    # Add a new line at the end of the month
    print("")

    # Return day of week for first day for the month after this one
    return (first_day + days) % NUM_DAYS_IN_WEEK
```

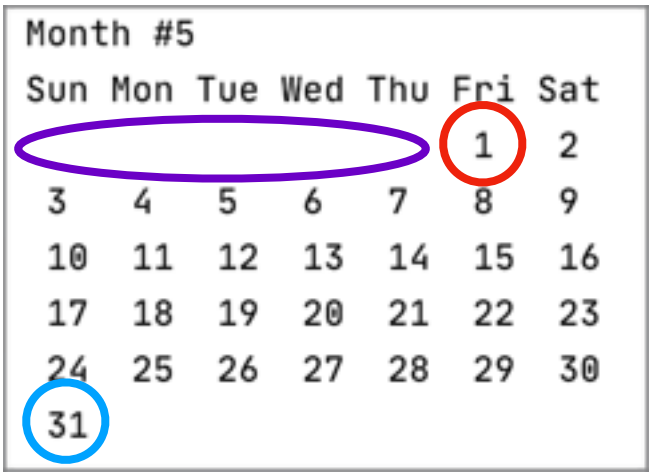
We'll go through this code step by step. I've marked off each section of the notes below to correspond to the code above by framing them in different colors.

```
print_month_header(month)
days = days_in_month(month, year)
```

- First, we print the value of `print_month_header()` created in **Step Four** with the value from `month` as the argument.
  - For example, the value of `month` is `5`, then when we use that in `print_month_header()` it will print:

```
Month #5
Sun Mon Tue Wed Thu Fri Sat
```

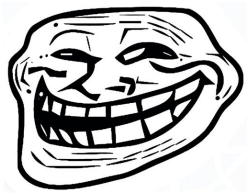
- Then, we create a variable called `days` and set its value by calling `days_in_month()` created in **Step Three** with the values from `month` and `year` as the arguments.
  - Recall `days_in_month()` will return the number of days in the month.
    - For example, if `month` is `5`, then `days_in_month()` will return `31` since there are 31 days in the month of May.
    - In the example to the right, we see that the *last* day of Month #5 is Sunday, so we know the 1st day of Month #6 will be Monday (so `first_day = 1` (it's zero-indexed.)



  - Next, we have to **print the space that leads up to the first day of the month**.
    - Notice above we have some white space to get to the 1st of the month. We'll need to fill in some spaces.
  - We use a `for` loop to accomplish this:

```
# Print leading space before first day in this month
for i in range(first_day):
  print("    ", end="") #prints four spaces per day
```

- In this block of code, `i` is set to whatever the value of `first_day` is. The loop will print four spaces per day, and we make sure it all stays on the same line with the code `end=“”`.
- Continuing our example, since `first_day` has a value of `5` in Month #5, the `for` loop will execute 5 times, printing a total of 20 spaces. This gets the number to line up with the day.



- Next up in this function (this is a really long function!) — insert bad joke here…
- We need to print numbers for all the days in the month! Here’s the code we use to accomplish this:
- We use a `for` loop once again, and set the range to *begin* at `0`, and end at the value derived from our `days` variable (we created in the first part of this function.)

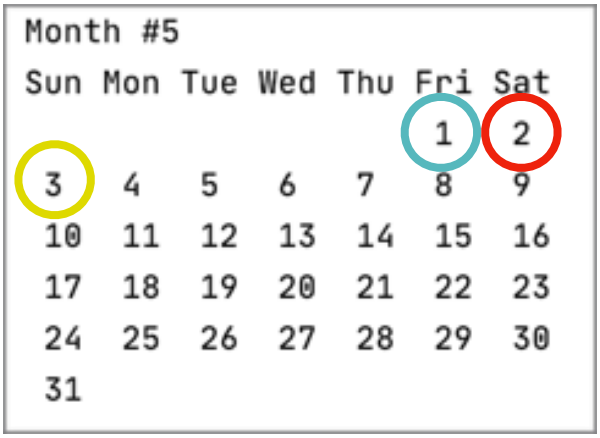
```
# Print numbers for all the days in the month
for i in range(0, days):
  print(format_number(i + 1), end="")
```

- In our `for` loop, we have a `print` statement. We use `i + 1` because since `i` begins at `0`, *we don’t want `0` printing as a day of the month!* So when `i = 0`, `i + 1 = 1`.
- We put that into `format_number()` created in **Step Five,** which will check if the number contains one or two digits. Remember `format_number()` will add an extra space if there is only one digit.
- Finally, we see the end of the print statement contains `end=“”`, which means all of the prints will print on the same line.

- Our next piece of code will add a new line at the end of the week:

```
# Add a new line at end of the week
if ((first_day + i) % NUM_DAYS_IN_WEEK) == (NUM_DAYS_IN_WEEK - 1):
  print(“”)
```

- We have an `if` statement that checks if **two values** are equal to each other. If they are, that means we're at the end of the line and we need to do a carriage return.
  - The **first** value in the `if` statement is the result of `(first_day + i) % NUM_DAYS_IN_WEEK)`. This equation will help us check if we've reached the end of the week and need a line break.
  - The **second** value is the result of `(NUM_DAYS_IN_WEEK - 1)`. This equation tells us *where* the end of the week is.
    - We have saved the value of `7` into our constant `NUM_DAYS_IN_WEEK` because there are 7 days in a week.
    - Because `i` starts at zero, we have to subtract 1 from our constant



`i` starts at `0` for the 1st day…

So for the 2nd day, `i = 1` here

•

```
    # Add a new line at the end of the month
    print("")
```

```
# Return day of week for first day for the month after
this one
return (first_day + days) % NUM_DAYS_IN_WEEK
```

- to compare the two sides of the equation.
- Therefore, the second part of our equation has the value of `6`

```
def first_day_of_year(year):
    """
    Returns the first day of the week for a given year,
    where (0 = Sunday, 1 = Monday, ..., 6 = Saturday)
    >>> first_day_of_year(2020)
    3
    """
    year -= 1
    return (year + (year // 4) - (year // 100) +
      (year // 400) + 1) % NUM_DAYS_IN_WEEK
```

because 7 - 1 = 6.

- If these two numbers are the same, then our Terminal will do a carriage return (`print("")`).

LET'S TRY AN EXAMPLE (because this code got me like ????)

- Recall the value of `first_day` is `5` (the 1st begins on Friday.)
- If we are trying to create a line break on the 2nd (Saturday), `i = 1`

```
def main():
    """
    Prints calendar for the year entered by the user
    """
    year = int(input("Enter year for calendar: "))
    first_day = first_day_of_year(year)

    # Loop through months 1 through 12
    for month in range(1, NUM_MONTHS + 1):
        first_day = print_month(first_day, month, year)
```

when we print the 2nd day. (Since `i = 0` at the 1st day.)

  - Therefore, `(first_day + i)` can be interpreted in this instance as `(5 + 1)` which gives us the value of `6`.
  - `6 % 7 = 6` since 6 *never* fits into 7, so its remainder is 6. **The first value on the left side of our code in this instance is `6`.**
- Recall that the second value on the right side of our code is `6`. Our `if` statement says if the left side and the right side are the same, we should print a line break.
- In this example, the left side evaluated to `6` and the right side evaluated to `6`. Therefore, `print("")` will happen.
- Sure enough, we have a line break and 3 is printed on the next line!

- The last part of this code block will print a new line at the end of the month. (Phew! Freakin finally):
- And finally, our return statement! This piece of code will return the day of the week for the 1st day in the month *after* this one.
- In the example with Month #5, the piece of information that will be returned is the following:
  - `(5 + 31) % 7`
  - `(36) % 7`
  - `1`
- As you can see, the value returned, `1`, is **where the next month's 1st day will begin.**

Step Seven: Print Calendar for a Given Month & Year.

- This function will help our program figure out the first day of the year.
- This code was taken from http://mathforum.org, so for further explanation, please visit the website.

Step Eight: Write main() function.

- Finally, we can put it all together! This function, `main()`, prints a calendar for the year entered by the user.
- Our code will first accept an input by prompting the user, "`Enter year for calendar:` ".
- Once the year is accepted, our variable `first_day` from **Step 6** will call the function `first_day_of_year` and submit the user's input of year as the argument.
- From there, the function `first_day_of_year` created in **Step 7** will run with the user's input, and will return the value for the first day of the year.
- Finally, we have a `for` loop that will start with `1` as the first month (representing January) and continue until `NUM_MONTHS + 1` (there are 12 months in a year, so `NUM_MONTHS = 12`) — because remember, ranges are *not* inclusive of the top number.
  - That loop will set the value of `first_day` as the value of `print_month()` created in **Step 5** with arguments of `first_day`, `month`, and `year` passed in.
  - Recall that the `print_month()` function:
    - prints a daily calendar for a given month and year,
    - determines the day the month begins on, and
    - returns the day for the first of the *following* month.
- This `for` loop will run for every month of the year (so it will run 12 times, once for every month.)


THAT'S IT!!!
I NEED A DRINK

CS106A Notes - Python
Week 3 Notes: Images (5/1/2020 Lecture)

Recap on 4/29/2020 Lecture

STYLING

- **Global variables** are variables that are visible to all functions.
  - These are different from **constants**, which are also visible to all functions but stay the same no matter what.
  - We don't use global variables because it's not good style!
  - When we declare global variables *and also* create new variables inside another function, the naming can get tricky and confusing.
- Here's an example of global variables:

```
NUM_DAYS_IN_WEEK = 7

balance = 0

def main():
  balance = int(input("Initial balance: "))
  while True:
    amount = int(input("Deposit (0 to quit): "))
    if amount == 0:
      break
    deposit(amount)

def deposit(amount):
  balance += amount
```

This is a `constant` visible to all functions (but always stays the same)

`balance` is a global variable visible to all functions, but it can change.

`balance` in the global scope is a **DIFFERENT** variable from `balance` here, but *it has the same name*. WHAT?!?!

This function wants to update `balance` in `main()` with the new amount

- Gee, how confusing. Can we make it cleaner?

```
def main():
  balance = int(input("Initial balance: "))
  while True:
    amount = int(input("Deposit (0 to quit): "))
    if amount == 0:
      break
    balance = deposit(balance, amount)

def deposit(balance, amount)
  balance += amount
  return balance
```

`balance` is declared here and assigned a value of whatever the user types in

`amount` takes input on *how much* user wants to deposit

Here, we're *updating the value* of `balance`

`deposit()` is called and is passed copies of `balance` and `amount`.

`deposit()` runs and adds `amount` to `balance`.

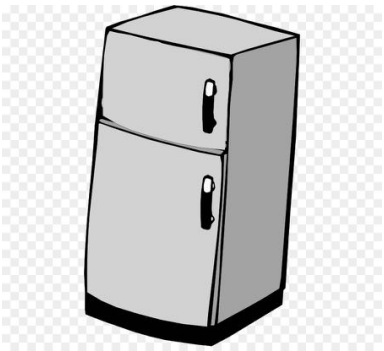
Finally, `balance` is returned with the updated value.

- **Parameters!** Parameters make for good styling. :)

- **ENCAPSULATION PRINCIPLE: All data used by a function should either be an argument sent to a parameter, or encapsulated in the function.**
  - It should be either a local variable inside the function, or an argument that goes into the function's parameters.

WHAT ABOUT MY TOAST?

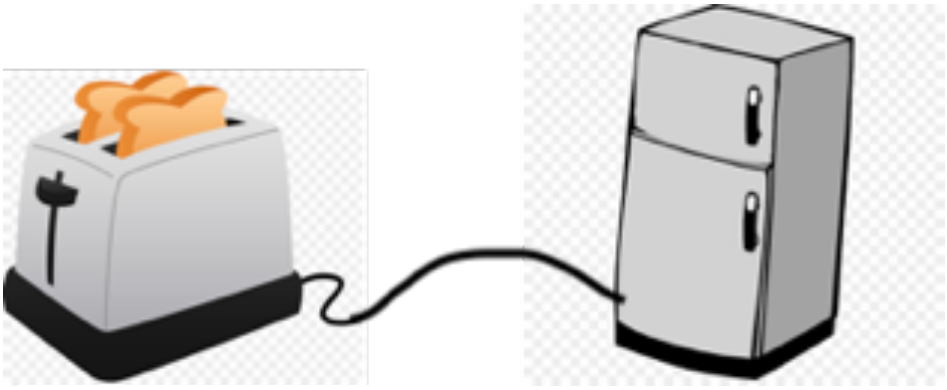
- Remember our toaster analogy? If we want some toast, we want to:
  - Take the bread out of the refrigerator

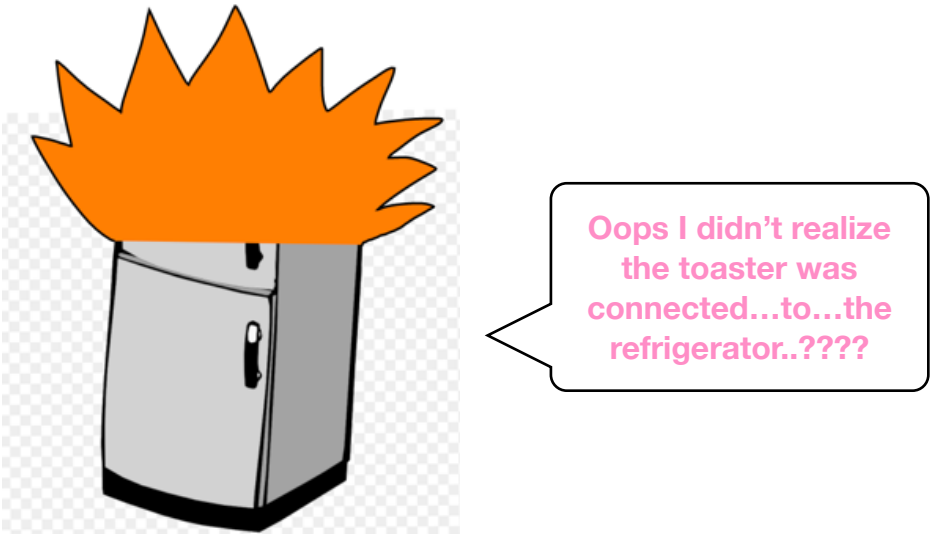


  - Put the bread into the toaster

  - Have delicious, warm toast (mmm toast)

- But what if you put the bread into the toaster, pushed the plunger down to toast your bread, and then ***your refrigerator caught on fire?????***

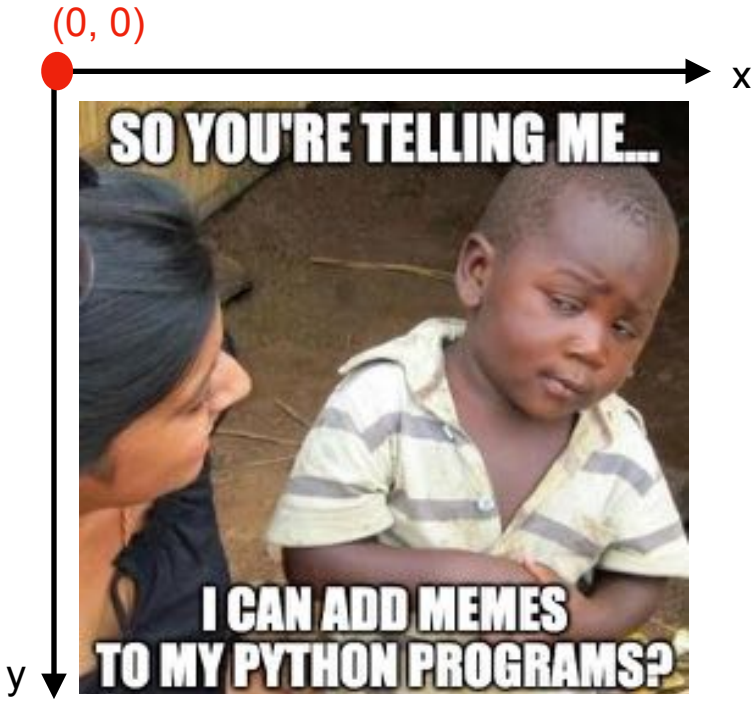




- It's a "no" from me.
- It's a "no" from Mehran.
- It's a "no" from Simon Cowell.
- So…just don't use global variables.

Notes by Regina Gerbeaux

Images in Python



INTRODUCTION TO IMAGES

- An image is made up of **pixels**. **Pixels are little squares that have individual colors associate with them.**
  - Every pixel has an (x , y) coordinate in the image
  - Origin (0, 0) is in the upper-lefthand corner.
    - y *increases* by going down
    - x *increases* by going right
- Pixels have 3 **RGB** value (red-green-blue values)
  - You can make any color using RGB (for example, mixing red and green will give you yellow.)
  - RGB has a value from 0 to 255.
  - Every value represents brightness for that color.

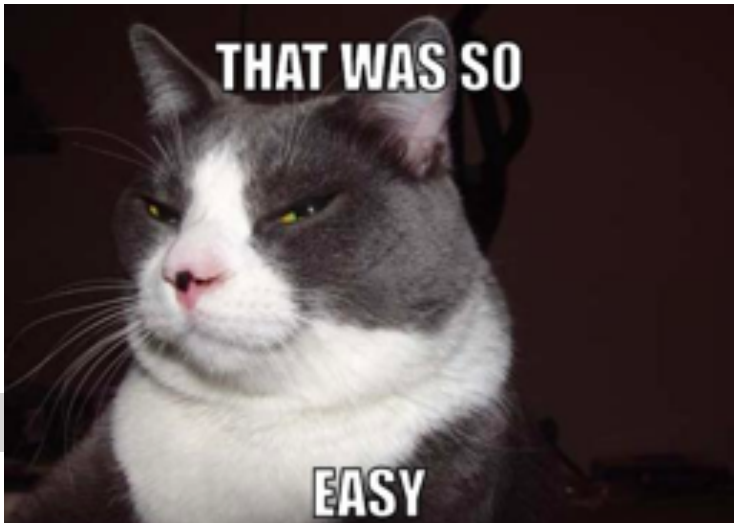
SHOW ME SOME ART

- If we zoom in on an image, we can actually see pixels (it'saaaahhh meeeee, Marioooo)
- You can identify the pixels by referencing coordinates (for example, (7, 1) is a black px.)
- Different colors have different RGB values.
  - For example, looking at Mario at (7, 1), our RGB value is R: 0 G: 0 B: 0 because black has no R, G, or B.
  - If we look at (7, 4), the RGB value is R: 255 G: 51 B: 51 and red has the greatest intensity.

Ahh, yes…the good ol days of Microsoft Paint. We didn't even realize we were playing with pixels as children.

HOW DO I MAKE MY OWN DANK PIXEL ART?

- We need to use the Python Imaging Library (PIL) - there's a version called *Pillow* (Nick Parlante built a library called **SimpleImage** using Pillow) and that's what we use in this class.
- Make sure you install Pillow first so we can use SimpleImage!
- To install, go into your Terminal on computer or Pycharm, and type either `py -m pip install Pillow` (PC) or `python3 -m pip install Pillow` (Mac). Tada!
- Reference Handout #8 for more info



HOW TO IMPORT

- To use SimpleImage, we have to **import** it into our Python file.
- There is a file called `simpleimage.py` which contains the SimpleImage library. To use it, import it into your program file by typing `from simpleimage import SimpleImage`
  - That's the format to import modules: `from location import module`.
- We've previously imported stuff into our files, but we typically do a direct import. When we import this way, we have to prepend our code with the file location along with the function we're calling.

```
import random

def main():
  number = random.randint(0, 9)
```

We imported the **entire** `random` file here.

Notice to call the `randint()` function, we have to prepend it with `random` so our file knows where to look for that function.

```
import simpleimage from SimpleImage
```

We imported the `SimpleImage` **module** from the `simpleimage` file

- By citing the location along with the specific module we want access to, we don't have to import *everything* in that file. When we import this way, we don't have to prepend our code with the file location.

FUNCTIONS IN THE SIMPLEIMAGE LIBRARY

- SimpleImage functions include the following things.
  - We can **create a SimpleImage object** by **loading an image from a file and storing it in a variable.**
    - Acceptable file types include jpg, png, and gif
    - SimpleImage objects are made up of Pixel objects
    - To create a SimpleImage object, follow this format:

```
variable = SimpleImage(filename)
```

    - You create a variable name, and set its value to `SimpleImage` along with the name of the picture you're referring to.
    - Notice the capitalization in `SimpleImage`; this is *not* using snake_case.
  - We can **display the image** on your computer.
    - To show the image, follow this:

```
variable_name.show()
```

  - We can **manipulate the image** by changing its individual pixels
  - You can also **create new images** (by getting a blank image) and set its individual pixel values

ACCESSING PIXELS IN AN IMAGE

- Recall from above that we can *create or manipulate* images by *setting its individual pixel values.* To do that, we have to access the pixels in an image.
- We can do this by using a new kind of loop called the `for-each` loop.
- Recall what the `for` loop looks like: it has an index (`i`), specifies how many times it will repeat (`in range(num)`), and then says what it will repeat doing.

```
for i in range(num):
  # do something num times
```

HeY iTz OuR fRiEnD DaH FoR LoOp~!!

Yo I'm `for-each`

```
for item in collection:
  # do something with item
```

Woahhhh what?

- The `for-each` loop looks like this:
- Instead of specifying the *number of times* something should repeat (as it does in the `for` loop), the `for-each` loop specifies that for *every item in the collection*, something should repeat.

- To reiterate, **the `for` loop says, "Loop [this] many times." The `for-each` loop says, "Loop for every item in this collection."**
- The reason why we use `for-each` here is because an image is made up of pixels, so images are *collections* of pixels.
- Here's an example:

```
mario = SimpleImage("mario.jpg")
for pixel in image:
  # do something with those pixels
```

This is the SimpleImage object. We set the value of `mario` to this object.

This is our `for-each` loop. `pixel` is the item and `image` is the collection.

This code gets repeated with each `pixel` in the image.

PROPERTIES OF IMAGES AND PIXELS

- **Properties of Images**
  - **Get the width and height** (values are returned in pixels) - use `image.width` and `image.height`
- **Properties of Pixels**
  - **Get the x, y coordinates** - use `pixel.x` and `pixel.y`
  - **Get the RGB values** (returned as integers between 0 and 255) - use `pixel.red`, `pixel.green`, and `pixel.blue`
    - Higher R, G, or B values = more of that color in the pixel
  - **Set RGB values to change them** (`.set_pixel()`)

Notes by Regina Gerbeaux

First Examples of Images & Pixels

CODE DAMMIT: DARKER IMAGE & RED CHANNEL EXAMPLE

- The following code will make an image darker. Let's go!

```
def darker(filename):
  """
  Reads image from file specified by filename.
  Makes image darker by halving RGB values.
  Returns darker version of image.
  """

  image = SimpleImage(filename)
  for pixel in image:
    pixel.red = pixel.red // 2
    pixel.green = pixel.green // 2
    pixel.blue = pixel.blue // 2
  return image
```

`darker` is a function that has the parameter `filename`. (Using yellow here to be ironic, harhar)

`image` is a variable whose value is the SimpleImage object of `filename`.

`for-each` will loop through every pixel in the image and half its R, G, and B pixel values to make it darker.

It returns the updated `image` once that's done.

- Meanwhile, this code will get a red channel (the intensity of red in an image) - creates a red version of the image

```
def red_channel(filename):
    """
    Reads image from file specified by filename.
    Changes the image as follows:
    For every pixel, set green and blue values to 0
    yielding the red channel.
    Return the changed image.
    """
    image = SimpleImage(filename)
    for pixel in image:
        pixel.green = 0
        pixel.blue = 0
    return image
```

`red_channel` is a function that has the parameter `filename`.

`image` is a variable whose value is the SimpleImage object of `filename`.

`for-each` will loop through every pixel in the image and update the G and B values to 0.

It returns the updated `image` once that's done.

- Now that we have these two functions, we can call them in main:

```
def main():
    original_flower = SimpleImage('images/flower.png')
    original_flower.show()

    dark_flower = darker('images/flower.png')
    dark_flower.show()

    red_flower = red_channel('images/flower.png')
    red_flower.show()
```

This creates the SimpleImage object `original_flower`

`.show()` makes it appear when you run the program

This creates the SimpleImage object `dark_flower`. Its value calls `darker()` and its argument is `original_flower`.

This creates the SimpleImage object `red_flower`. Its value calls `red_channel()` and its argument is `original_flower`.

- And here's what the output looks like:

GreenScreen

INTRODUCTION TO GREEN SCREENS

- Now that we have the basics down of how to access and manipulate pixels on a screen, let's try something new: **Greenscreening**!
- **Greenscreening** can change a picture by manipulating its pixels
  - What you've seen in movies and Zoom backgrounds
  - We have an original image with pixels that are "green enough".
  - Then, the computer replaces those *green* pixels with pixels from corresponding (x, y) locations of another image.

HOW TO MAKE SOME DANK ~~*MEMES*~~ GREENSCREENS

- We have to specify something called an **intensity threshold**.
  - An `INTENSITY_THRESHOLD` is a constant that specifies *how* intense a green has to be to qualify as "green enough."
  - Recall our program will replace certain green pixels that are "green enough." `INTENSITY_THRESHOLD` helps determine that #
  - The `INTENSITY_THRESHOLD` works by accepting a float. The float specifies **how much more intense** a green pixel has to be from the average of its RGB values.
  - For example, if `INTENSITY_THRESHOLD` is 1.6 and the average intensity of *that* pixel's RGB values is 75, our function runs if the green pixel's value is >= 120.

- Take this for example:

```
INTENSITY_THRESHOLD = 1.6

def greenscreen(main_filename, back_filename):
    image = SimpleImage(main_filename)
    back = SimpleImage(back_filename)

    for pixel in image:
        average = (pixel.red + pixel.green + pixel.blue) // 3
        if pixel.green >= average * INTENSITY_THRESHOLD:
            x = pixel.x
            y = pixel.y
            image.set_pixel(x, y, back.get_pixel(x, y))
        return image
```

- Here, a pixel is categorized as "green enough" if it's 1.6x the avg intensity of the RGB values in that pixel (`INTENSITY_THRESHOLD`)
- We have a function called `greenscreen` that takes two parameters: `main_filename` and `back_filename`.
- We save `main_filename` as the value of **`image`** and `back_filename` as the value of **`back`** to turn them into **SimpleImage objects.**
- Then, we have a `for-each` loop that will access every pixel in `image` (our main image)
  - We define a variable named `average` and set its value to take the average of all pixel colors in `image`. (This takes all red, green, and blue pixels, and then does integer division on it to find the average.)
  - Then, we have an `if` statement that checks the current pixel's green value against the average of the RGB values in that current pixel multiplied by the intensity threshold.
    - If the current pixel's green value is greater or equal to the average * intensity, then we create x and y and set them equal to the current pixel's coordinates using pixel.x and pixel.y

  - Lastly, we access the main image (`image`) and call `.set_pixel()` to *manipulate its pixels*. It **overwrites the pixels in that position.**
    - In this function, we pass in `x` and `y` defined in the previous step, and override it with the corresponding `x` and `y` *from our background image* (`back`) using `.get_pixel()`!
- Once our `for-each` loop has run through every single `pixel`, it will return the new image to the calling function.

EXECUTING `GREENSCREEN.PY`

- In lecture, we change the function we wrote above to check **red pixels** instead of green.
- ***This is a note that we can also do bluescreens, or redscreens, in addition to greenscreens!***
- This is because the function works the same way — it just looks for different colors of pixels.
- Recall that our pixels are made up of R, G, and B, so the logic is the same.

- Here is our code to check for red pixels:

```
INTENSITY_THRESHOLD = 1.6

def redscreen(main_filename, back_filename):
    image = SimpleImage(main_filename)
    back = SimpleImage(back_filename)

    for pixel in image:
        average = (pixel.red + pixel.green + pixel.blue) // 3
        if pixel.red >= average * INTENSITY_THRESHOLD:
            x = pixel.x
            y = pixel.y
            image.set_pixel(x, y, back.get_pixel(x, y))
        return image
```

- If it looks familiar, that's because all I did was change the function name (`redscreen` instead of `greenscreen`) and change the `if` condition from `pixel.green` to `pixel.red`.
- How do we run it? Here's what our `main()` looks like:

```
def main():
    original_stop = SimpleImage('images/stop.png')
    original_stop.show()

    original_leaves = SimpleImage('images/leaves.png')
    original_leaves.show()

    stop_leaves_replaced = redscreen('images/stop.png',
'images/leaves.png')
    stop_leaves_replaced.show()
```

- Here, we've created three SimpleImage objects: `original_stop`, `original_leaves`, and `stop_leaves_replaced`.
  - `original_stop` is a photo of a stop sign.
  - `original_leaves` is a photo of leaves.

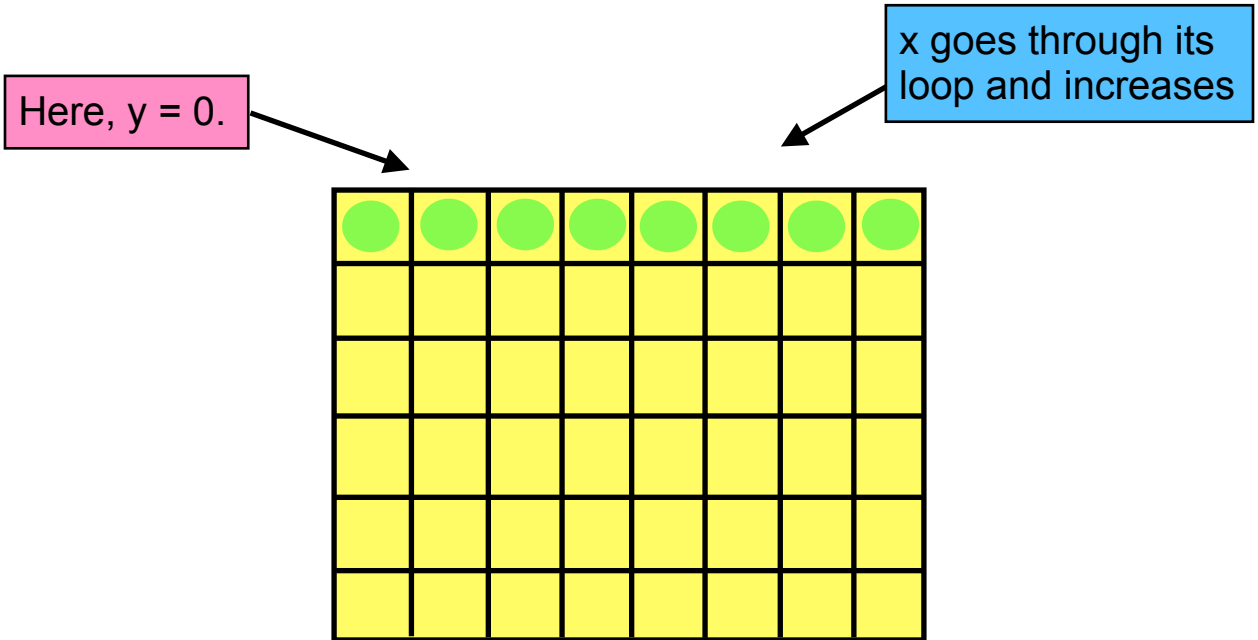
    - `stop_leaves_replaced` calls the `redscreen()` function and provides our stop image as the `main` image, and our leaves image as the `back` image.
- `redscreen()` runs and replaces all of the “red enough” `pixels` in the Stop image with the corresponding Leaves image, and `returns` the new photo.
- Lastly, notice we have `.show()` called on all three images. This will **display** all three images (the originals of the stop image and the leaves image, and the manipulated image.)
- Here are the images:

Mirrored (Mirroring an Image)

INTRODUCTION TO NESTED FOR LOOPS

- We can create mirrored images using **nested `for` loops. Nested `for` loops are loops within loops.**
- Remember that **images are just a bunch of pixels going left to right (x axis) and top to bottom (y axis).**
  - If we want to do something to an image (such as mirroring it), we can *use nested `for` loops to accomplish this!*
- Here's an example:

```
image = SimpleImage(filename)
width = image.width
height = image.height

for y in range(height):
  for x in range(width):
    pixel = image.get_pixel(x, y)
    # do something with pixel
```



- In the above example, `y` will stay constant while `x` finishes running all its loops. **This is because nested `for` loops run the inside loops first until they all finish *before* running outside loops**.
- So when we first enter this nested `for` loop, `y = 0`. Great, then we go into the second loop. There, `x = 0` first, then `x = 1`, then `x = 2`, and so on…all while `y` remains at `0`. All of these pixels' coordinates are retrieved with `.get_pixel()` and stored in `pixel`
- Once `x` finishes going through all its loops, this loop *exits* and goes back to `y`. Now, `y = 1`! And just as you might have guessed, `x = 0`, then `x = 1`, then `x = 2`…and so on.
- This loop will continue until the entire picture is covered.
- **Note**: x and y are zero-indexed, so if you have an image whose width is 100 pixels and whose height is 50 pixels, its x coordinates go from 0 to 99, and its y coordinates go from 0 to 49.

MIRRORING AN IMAGE: DECOMPOSITION

- If we think about it, mirroring an image is taking an image and duplicating it with the x coordinates reversed.
  - e.g.: If an image is 50px wide, then the x axis goes from 0 to 49.
  - When we mirror the image, the coordinate at (49, y) should now be (50, y); the coordinate at (48, y) should now be (51, y); etc.

Harhar, because this is kind of like a mirror…kind of…? Never mind.

- Our y axis remains unchanged, since we aren't manipulating the image to be upside down.

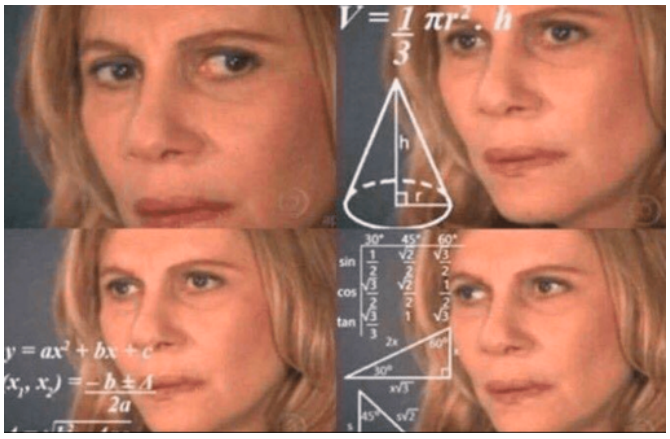
MIRRORING AN IMAGE: CODE IT

Mirror Image Function

```
def mirror_image(filename):
  image = SimpleImage(filename)
  width = image.width
  height = image.height

  # create new image to contain mirror reflection
  # save it to a new variable called mirror
  mirror = SimpleImage.blank(width * 2, height)

  for y in range(height):
    for x in range(width):
      pixel = image.get_pixel(x, y)
      # set pixels of original image
      mirror.set_pixel(x, y, pixel)
      # now set mirrored pixels
      mirror.set_pixel((width * 2) - (x + 1), y, pixel)
  return mirror
```

- First, within `mirror_image( )` passing in an image file, we create a variable called `image` and set its value to be the SimpleImage object of our imported filename.
- Then, we create variables `width` and `height`, and set their values to be the image's width and the image's height respectively, using `image.width` and `image.height`.
- Next, we create a variable called `mirror`, and set its value to be a *new SimpleImage object*. This image is blank and we pass in its dimensions as arguments.

    - The reason why `width` in this case is `width * 2` is because the width has to be big enough to hold *both* the original and the mirrored images all in one image.
- Now, we have a nested `for` loop as seen before.
    - We go through every pixel in the image, and save those pixels using `.get_pixel()` to a new variable called `pixel`.
    - The loop will access each pixel left to right starting at the topmost row, and working its way down to the bottom of the image.
    - As it retrieves the pixels, the loop sets those pixels in our new image. First, we set the pixels of our *original* image since we're copying it into the new image.
        - We pass into `mirror.set_pixel()` arguments of `x`, `y`, and `pixel`.
            - In our *new* image, the pixel located at the same coordinate is the <u>same</u> as the `pixel` at the current coordinate in our *original* image.
            - For example, the pixel at (2, 4) in our *original image* will be the same as the pixel at (2, 4) in our *new* image.
    - Then, we have to set the pixels of our *mirrored* image! We pass into `mirror.set_pixel()` the following arguments:
        - In our *new* image, the pixel located at `(width * 2) - (x + 1)` in the x direction is the <u>same</u> as the current `pixel`'s `x` direction in our *original* image.
            - For example, if x = 30 in our *original* image and the original image's width is 30px, x = 31 in our mirrored image.



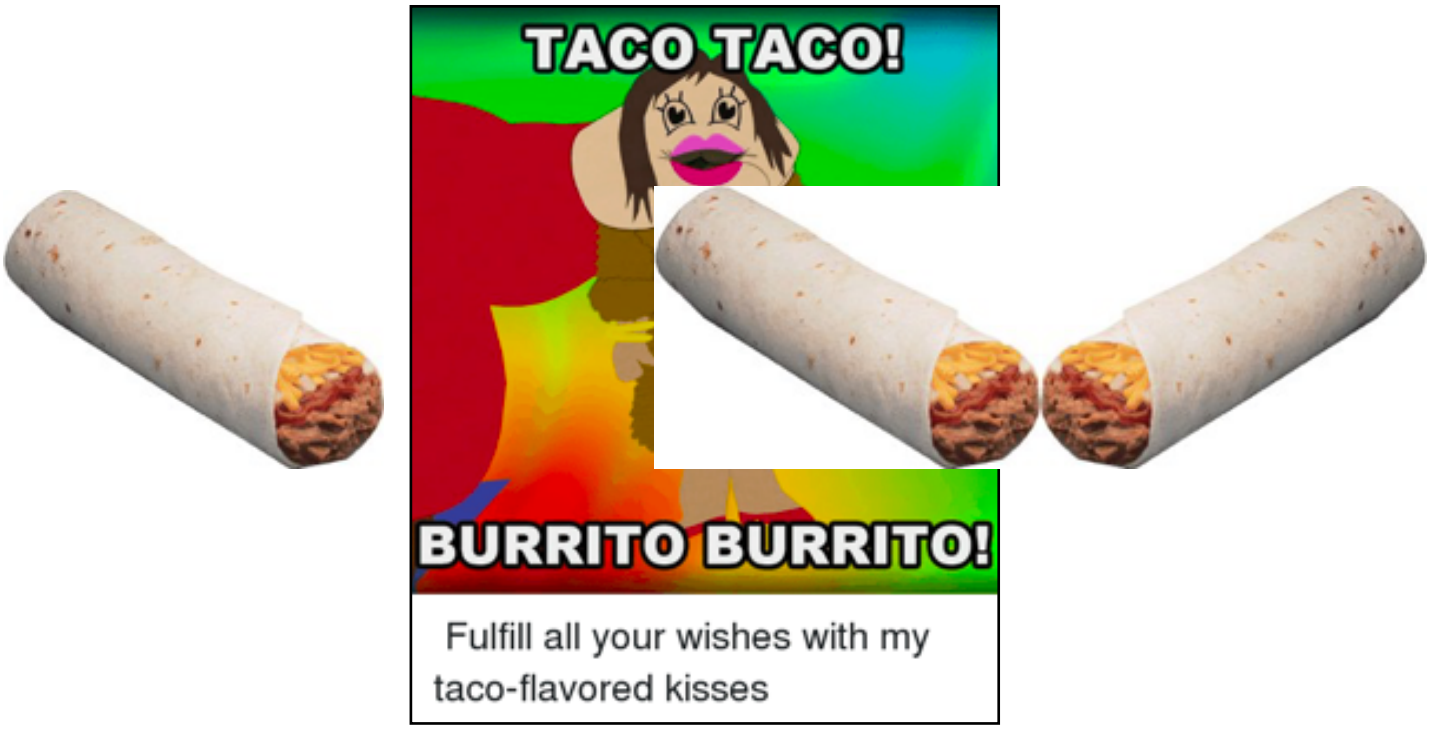
Actual image of me doing the math for this part.

  - The y direction in the *new* image is the same as the y coordinate in the *original* image.
- The last thing we do in this function is return the value of `mirror`.

Main Function

```
def main( ):
    original = SimpleImage('images/burrito.jpg')
    original.show( )

    mirrored = mirror_image('images/burrito.jpg')
    mirrored.show( )
```

- Create a variable called `original`, and set its value to be a SimpleImage object of our burrito from the **location of the file**.
- Create a variable called `mirrored` and set its value to call the function from above, `mirror_image( )`. We pass in the location of our amazing burrito, and let it do its magic. The function returns the mirrored image of our burrito. (So we get not one burrito, but TWO.)
- View your burritos by using `.show( )` appended to `original` for the original, and appended to mirrored for the mirrored version.



Differences Between `for-each` Loops and Nested `for` Loops

- If we compare the two following pieces of code (left = `for-each` loop; right = nested `for` loop), they accomplish the same thing:

```
for pixel in image:
  pixel.red = pixel.red // 2
  pixel.green = pixel.green // 2
  px.blue = pixel.blue // 2
return image
```

```
for y in range(image.height):
  for x in range(image.width):
    pixel = image.get_pixel(x, y)
    pixel.red = pixel.red // 2
    pixel.green = pixel.green // 2
    pixel.blue = pixel.blue // 2
```

- We use **nested for-loops** when we care about `(x, y)` locations. For example, we care about the locations of the `x y` coordinates when creating a mirrored image, so we used a nested `for` loop there.
- Use `for-each` when you don't care about the location.

Notes by Regina Gerbeaux

CS106A Notes - Python

Week 4 Notes: Graphics (5/6/2020 Lecture)

Recap on 5/1/2020 Lecture

IMAGES

- **Images** give us another chance to practice loops and parameters
- **Stack Overflow** is a great starting point to look for information for a problem you want to solve
  - Search for your topic (e.g.: "How to create a Sepia filter") on Stack Overflow
  - Look at what other people have done!
- Here's an example of creating a Sepia filter with a `for-each` loop:

```
def main():
    image_name = input('enter an image name: ')
    image = SimpleImage('images/' + image_name)
    for pixel in image:
        sepia_pixel(pixel)
    image.show()

def sepia_pixel(pixel):
    R = pixel.red
    G = pixel.green
    B = pixel.blue
    pixel.red = 0.393 * R + 0.769 * G + 0.189 * B
    pixel.green = 0.349 * R + 0.686 * G + 0.168 * B
    pixel.blue = 0.272 * R + 0.534 * G + 0.131 * B
```

```
def main():
    image_name = input('enter an image name: ')
    image = SimpleImage('images/' + image_name)
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            sepia_pixel(pixel)
        image.show()
```

- In `main()`, create the variable `image_name`. Allow user to enter the name of the image they want to put a Sepia filter on.
- Create the variable `image` and set its value to create a SimpleImage object. The location of the image is the folder `'images/'` with `image` (user input from previous step.)
- Use **`for-each`** to loop through every `pixel` in `image`.
  - Within the loop, run `sepia_pixel()` and provide argument of current `pixel`
    - Here, we take an old RGB value and turn it into a new RGB value that represents the hues in a Sepia photo.
    - Create variables R, G, and B to save time from typing out `pixel.red`, `pixel.green`, and `pixel.blue` over and over again.

        - Then, we have some math that we got from Stack Overflow: these are just some magic numbers that turn our red, green, and blue `pixels` into browner tones to match the Sepia filter.
    - Lastly, `show` the new `image`!
- You can also do this with a `nested for` loop. `sepia_pixel()` works the same as before, so this is `main()`:

    - In our **`nested for` loop…**
        - First, the outer loop will start with the first `y` location and loop through all of `image`'s height.
            - Within that, the inner loop will start with the first `x` location and loop through all of `image`'s width.
                - Within that, we declare the variable `pixel` and set it to the current `(x, y)` coordinate
                - Then, we run `sepia_pixel()` on `pixel`
            - Inner loop continues until it reaches the end of the `x` points e.g.: (0, 0), (1, 0), (2, 0), (3, 0), (4, 0)…
            - Inner loop exits; outer loop runs and moves to next `y` point
            - Inner loop executes once again with the new `y` point. e.g.: (0, 1), (1, 1), (2, 1), (3, 1), (4, 1)…
        - This `nested for` loop will continue until it loops through all `y` points, thus looping through all of the pixels in the image.
    - Then, just as before, we `show` the new `image`.

We use `nested for` loops when we care about the specific (x, y) points (such as when we mirror an image.) Otherwise, it's simpler to use `for-each` loops.

Notes by Regina Gerbeaux

Blue Rectangle

SMALL NOTE, RE: FINAL PROJECT

- Graphics are pretty simple if you have PyCharm installed on your computer. The graphic library we used is called **tkinter** (https://docs.python.org/3/library/tk.html)
- If you are using an Online IDE, it's hard to get graphics to work properly in the browser, or they'll work slower.
- Keep this in mind while building your final project, depending on where you're building it. :)

INTRO TO GRAPHICS PROGRAMS

- So far, we've learned Karel Programs and Console Programs - now, it's time to learn **Graphics Programs**! *Graphics programs are programs that have a graphical output.*
- Remember, for CS106A, the reason why we're learning graphics programs is because it's yet another opportunity to practice loops, functions, and passing parameters.

DRAW A SQUARE

- Let's jump into writing our first function to produce a graphic. Let's draw a **blue square**! Here's the code to do so:

```
def main():
  canvas = make_canvas(800, 200, 'Hello Rect')
  canvas.create_rectangle(20, 20, 100, 100, fill="blue")
  canvas.mainloop()
```

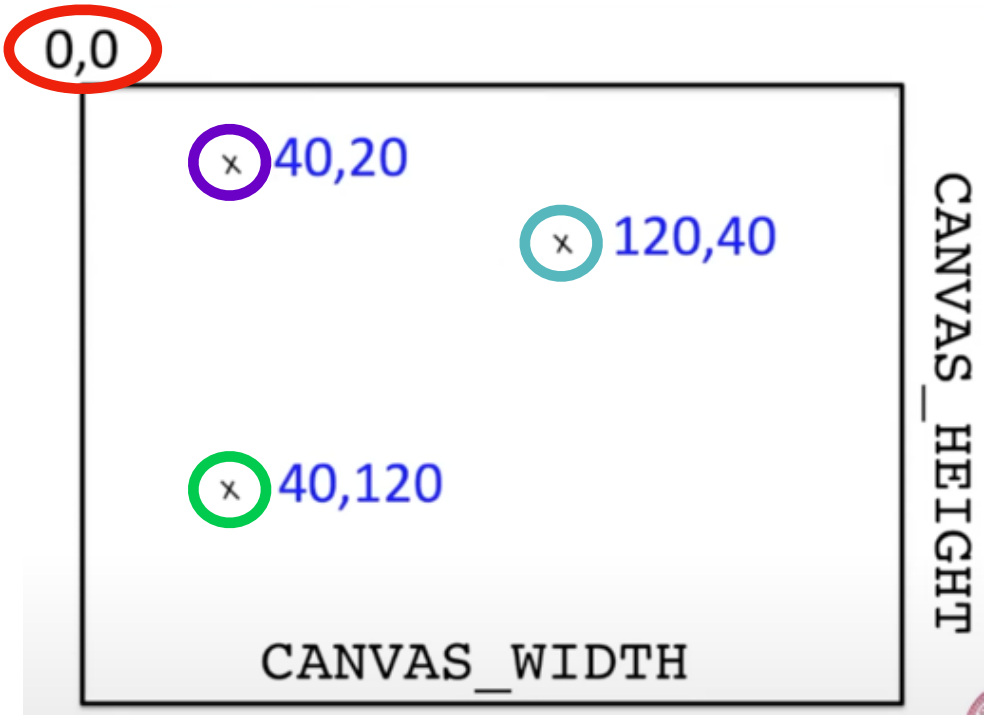
- First, we create a variable called `canvas`.
  - The value of `canvas` is the `make_canvas()` function, which creates a new, blank canvas.
  - Within `make_canvas()`, we set our `canvas`'s dimensions so it is `800px` wide and `200px` tall. We also name the file `Hello Rect`.
- Then, we create a rectangle using `canvas.create_rectangle()`.
  - First, we set the (x, y) **coordinates** for the location of the ***top-lefthand corner*** of the square. Here, we set the top-lefthand corner of the square to be at **`(20, 20)`**
  - Then, we set the (x, y) **coordinates** for the location of the ***bottom-righthand corner*** of the square. Here, we set the bottom-righthand corner of the square to be at **`(100, 100)`**
  - Lastly, we **fill** the square! We've set **`blue`** as the color.

Small Note about `fill="blue"`: Why do we have `fill=""`? This is the first time we have a **named argument** passed. And what **named arguments** do is they *allow for default values*. So, if you don't *want* to pass in that parameter, you don't have to! Its default value would be white.

- Then, we have `canvas.mainloop()`.
  - What this line does is continues keeping your new canvas open.
  - If you don't have this line, it will create your canvas, create your rectangle, and *destroy it before you even get to see it.* :(
  - `canvas.mainloop()` simply keeps the `canvas` up and open to see.

GRAPHICS COORDINATES

- Remember that coordinates start in the **top lefthand corner**. **(0, 0)** marks the top lefthand corner of your canvas.

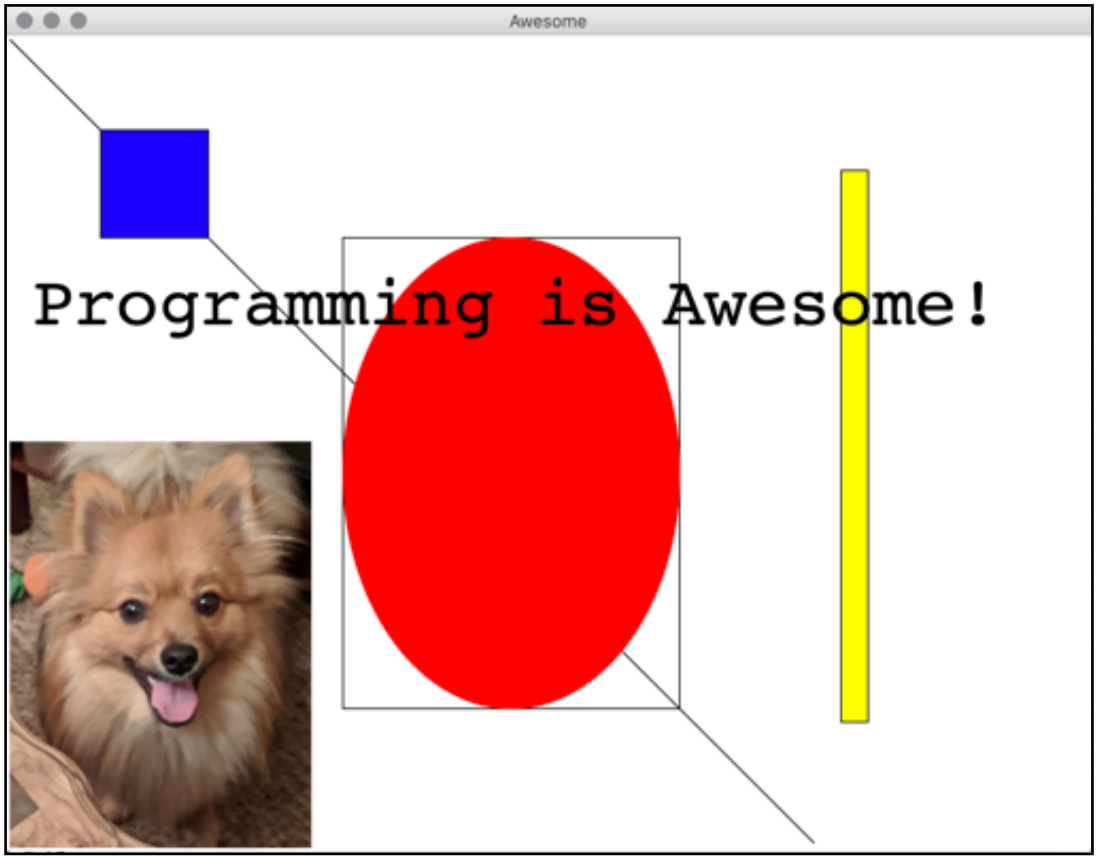


- Remember that x values are measured from the left, and y values are measured from the top.
  - **(40, 20)** is 40px from the left, and 20px from the top
  - **(40, 120)** is 40px from the left, and 120px from the top
  - **(120, 40)** is 120px from the left, and 40px from the top
- The key is to think __ pixels *from the left*, and __ pixels *from the top*.

Problem Solving: `programming_is_awesome.py`

RECTANGLES, OVALS, AND TEXT

- In `programming_is_awesome.py`, we create a canvas named `Awesome` with lots of cool graphics! Here's a photo of what that canvas looks like:



- Cool! Let's go line by line and see how we created this canvas.

- **Step 1**: Declare **width and height constants** for our canvas. Set `width` to `800px` and `height` to `600px`.

```
CANVAS_WIDTH = 800
CANVAS_HEIGHT = 600
```

- **Step 2**: Within `main()`, create **canvas** using `make_canvas()`
  - Use the constants declared above to set the new canvas's `width` to `800px` and its `height` to `600px`.
  - Name the canvas "Awesome".

```
def main():
    canvas = make_canvas(CANVAS_WIDTH, CANVAS_HEIGHT, 'Awesome')
```

- **Step 3**: Create a **line** going diagonally by setting its starting coordinate to `(0, 0)` and its ending coordinate to `(600, 600)`.

```
    # a line for good measure!
    canvas.create_line(0, 0, 600, 600)
```

- **Step 4**: Create a **blue square** with a width and height of 80px by:
  - Using `.create_rectangle()`
  - Specifying start coordinate of `(70, 70)`
  - Specifying end coordinate of `(150, 150)`
  - Setting `fill` to `blue`
- Because the coordinates start at `(70, 70)` and end at `(150, 150)`, there are `80px` in between; hence, our square is 80px

```
    # a blue square with width and height = 80
    canvas.create_rectangle(70, 70, 150, 150, fill="blue")
```

- **Step 5**: Create a **yellow rectangle** that is long and skinny by:
  - Using `.create_rectangle()`
  - Specifying start coordinate of `(620, 100)`

  - Specifying end coordinate of `(640, 510)`
  - Setting `fill` to `yellow`
- Because the x coordinates are 620 and 640 respectively, the width of our rectangle is going to be quite small: **only 20px wide**.
- Because the y coordinates are 100 and 510 respectively, the height of our rectangle is very long: **410px long**!

```
    # a yellow rectangle that is long and skinny
    canvas.create_rectangle(620, 100, 640, 510, fill="yellow")
```

- **Step 6**: Create a **red oval** and a **rectangle at the same spot**, by:
  - Specifying the start and end points for both shapes to be at the same place: `(250, 150)` and `(500, 500)` respectively
    - The way OVAL works is you specify the *rectangle* points, and the program will draw the biggest oval it can that fits within that rectangle.
  - Setting oval's `fill` and `outline` to be `red`
  - Notice rectangle doesn't contain a `fill` argument, so it will assume a color of white.

```
# a red oval and a rectangle at the exact same spot!
canvas.create_rectangle(250, 150, 500, 500)
canvas.create_oval(250, 150, 500, 500, fill="red", outline="red")
```

- **Step 7**: Make our program display an **image of Simba**! Awww! (Yes, you can also put images on your canvas!)
  - We use Pillow to create a Photo Image and save it to `image`
  - Type `.create_image( )` to create a new image! Within that:
    - Specify where the image begins `(0, 300)` and,
    - which part of the image is located there (an `anchor` refers to the part of the image that starts at the coordinate. In this case, it is the `northwest` ("`nw`") part.
    - Set the value of `image` to be `image` that we created before

```
# images require the pillow library
image = ImageTk.PhotoImage(Image.open("images/simba.png"))
canvas.create_image(0,300,anchor="nw",image=image)
```

- **Step 8**: Make our canvas display some **text**, by:
  - Using `.create_text( )` to create some text, within that:
    - Specify coordinates on where the text begins `(20, 200)`
    - Which part of the text is located there - the `west` part of the text is `anchored` at `(20, 200)`
    - the font name and size (`font = 'Courier 52'`)
    - Content of text (`text` reads `"Programming is awesome!"`)

```
# some text is written on the screen.
canvas.create_text(20, 200, anchor='w', font='Courier 52',
text='Programming is Awesome!')
```

—> ***Step 9*: "Dude, where's my rectangle?"**

- What if we wanted to create another rectangle as such?:

```
canvas.create_rectangle(0, 800, 10, 810)
```

  - The problem with this rectangle is that **the coordinates are not within the dimensions of our canvas!**
  - Recall that our canvas is `800px` by `600px` (from **Steps 1 & 2**.) However, notice above, our rectangle's y location is at `800` and `810`, respectively.
  - Therefore, this rectangle was created, but it's just off the canvas and therefore not visible.

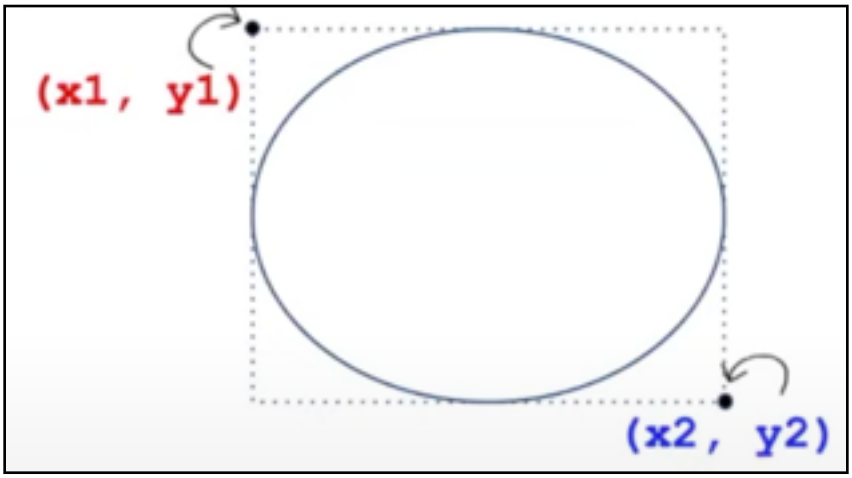
- **Step 10**: Make sure `canvas` stays open!
  - Recall that we need the line `canvas.mainloop()` in order for our `canvas` to always display. Without this line of code, `canvas` will generate and disappear before you can even see it!

```
canvas.mainloop()
```

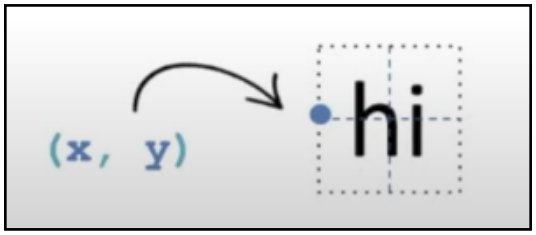
MAIN TAKEAWAYS

- To **create a line:**
  - Use `canvas.create_line()`
  - Provide four arguments of the coordinates for the **start** and **stopping** points (`x1`, `y1`, `x2`, `y2`)

- To **create an oval**:
  - Use `canvas.create_oval()`
  - Provide four arguments of the coordinates for a rectangle (`x1`, `y1`, `x2`, `y2`)
  - The program will generate the largest possible oval that can fit inside that rectangle.

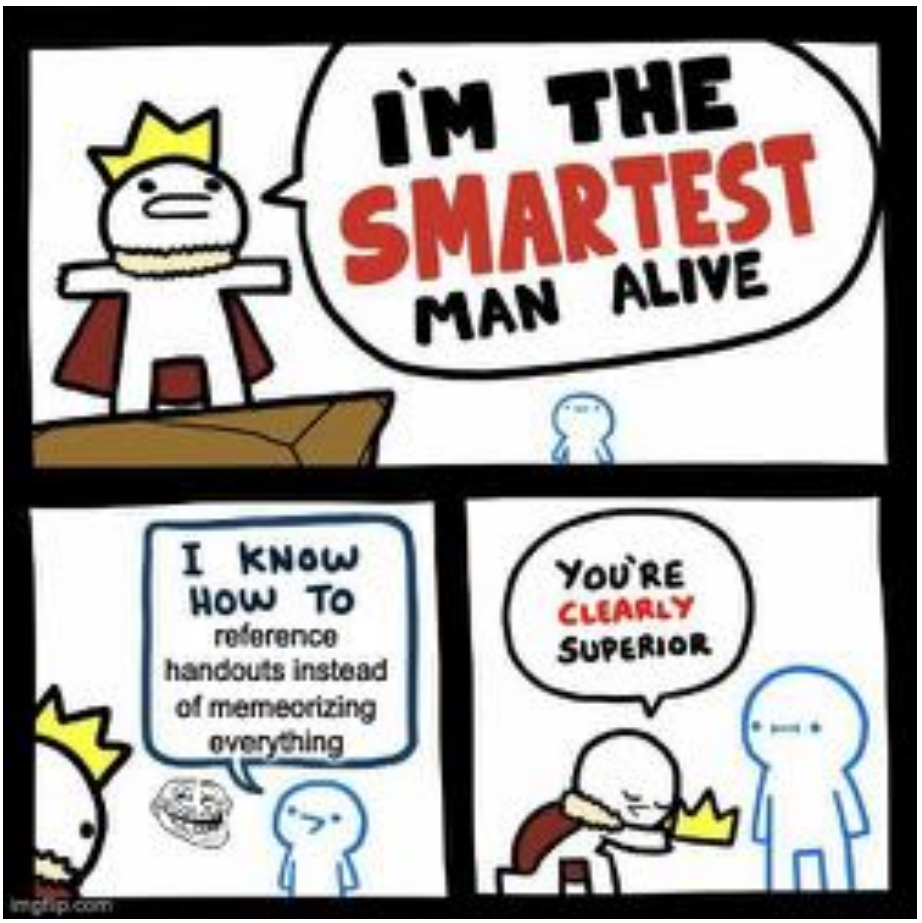


- To **create text:**
  - Use `canvas.create_oval()`
  - Provide these parameters:
    - The **location** of the text (`x` and `y`)
    - The `text` itself (e.g.: `'hi'`)
    - The `anchor` (which part of the text is at the `x y` coordinate) - e.g.: 'w' - stands for "west" which aligns the text to the Left



- For reference, go to https://compedu.stanford.edu/codeinplace/v1/#/handout/graphicsReference.html for the `tkinter` Graphics Reference handout.

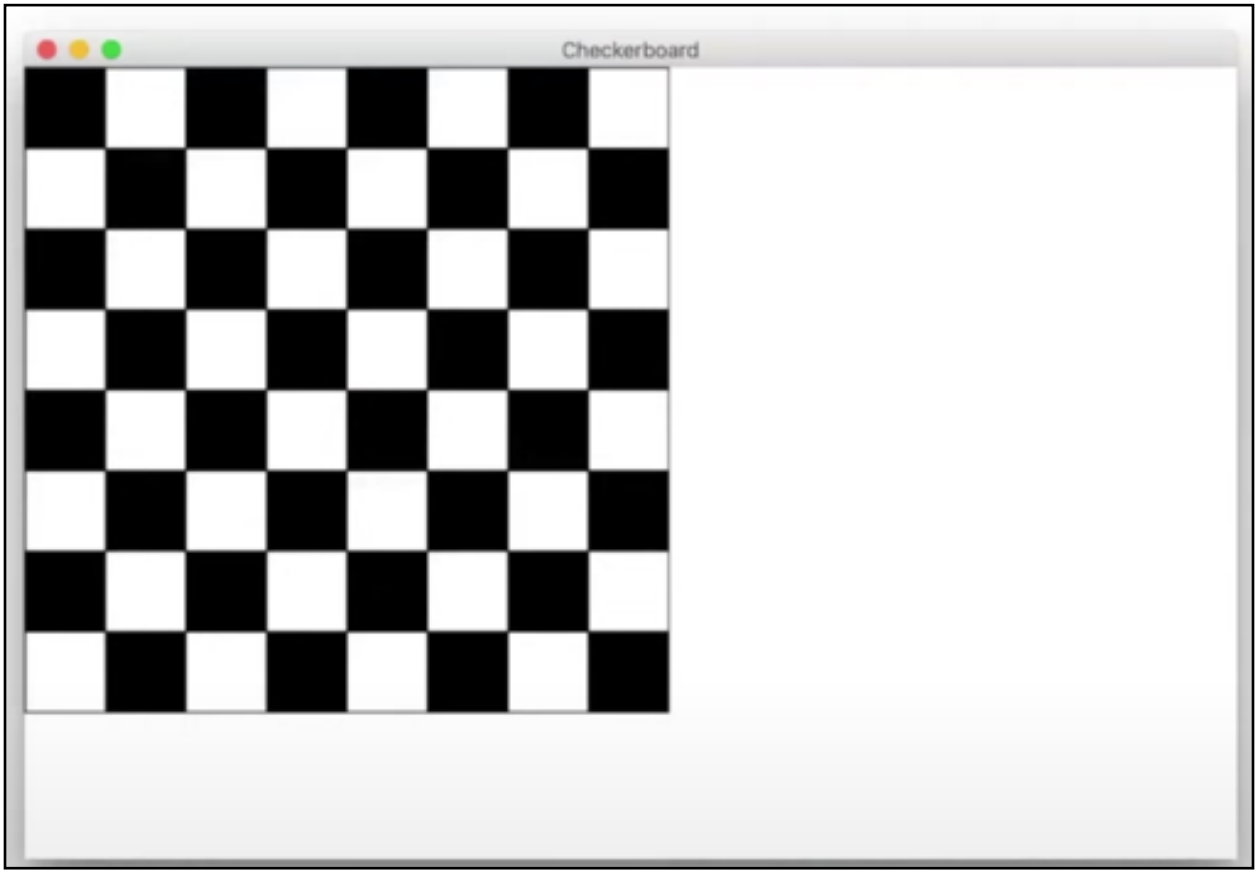
- Play around with the code above at https://compedu.stanford.edu/codeinplace/v1/#/example/programming-is-awesome.



Problem Solving: `checkers.py`

MAKE A CHECKERBOARD

- We're going to create a bunch of rectangles!



- One of the skills we're practicing in CS106A is writing loops, and this would be a really great time to practice doing that (instead of manually coding every rectangle on the board!)
  - Does this remind you a little bit of Karel? Instead of manually programming Karel to do stuff, we made Karel repeat something using loops.
  - This seems like a really good place to do this too!
- Start with **decomposing**.
  - Milestone 1A: Can you draw a **single square**?
  - Milestone 1B: Can you draw a **row of squares**?

- Milestone 2: Can you draw a **grid of squares**?
- Milestone 3: Can you **color the squares** to form a checkerboard?

PRE-MILESTONE: SETTING CODE UP

- We'll start off with this familiar starter code for graphics: *creating the canvas*.
  - Define constant for the canvas size. Since a checkerboard is a square, we only have to specify one number to serve as both our width and height. We'll make our `CANVAS_SIZE` variable `800px`.
  - We also create constants `N_ROWS` and `N_COLS` for number of rows in a chess board, and `SIZE` for the size of our squares.
  - Then, use `make_canvas( )` to create a canvas.
    - Specify the x and y coordinates to reference `CANVAS_SIZE`. In this case, our new canvas will be `800px x 800px`.
    - Name it '`Chess Board`'.

```
CANVAS_SIZE = 800

N_ROWS = 8
N_COLS = 8
SIZE = CANVAS_SIZE / N_ROWS - 1

# create the canvas
def main( ):
  canvas = make_canvas(CANVAS_SIZE, CANVAS_SIZE, 'Chess Board')
```

- Then, as you may recall, we always use `canvas.mainloop( )` at the end of our function to keep our canvas displaying.

```
canvas.mainloop( )
```

MILESTONE 1A: DRAW A SINGLE SQUARE

- Let's dive into the code that belongs in between the code we above!
- We have to draw a square, so it makes sense to write a function that can do this since we'll have to do a lot of them.

```
# draws one square
def draw_square(canvas, row, col):
  x = col * SIZE
  y = row * SIZE
  color = 'white'
  canvas.create_rectangle(x, y, x + SIZE, y + SIZE, fill=color)
```

- We create `draw_square()` and give it a parameter of `canvas`.
- Create variable `x` and assign the value to be `col` (current column number from `for each` loop below) multiplied by our `SIZE` constant (so the squares won't overlap.)
- Create variable `x` and assign the value to be `row` (current row number from `for each` loop below) multiplied by our `SIZE` constant (so the squares won't overlap.)
- Create variable `color` and set it to `white`.
- Then, we use `create_rectangle()` and pass in these params:
  - `(x, y)` to form the coordinate for the top left corner in our square — so the coordinate in this case is `(0, 0)`
  - `(x + SIZE, y + SIZE)` to form the coordinate for the bottom right corner in our square
    - `SIZE` is a constant we specified in our previous step, which takes `CANVAS_SIZE` (`800px`) and divides it by `N_ROW` (`8`) and subtracts `1`.
    - `800 / 8 = 100`, and `100 - 1 = 99`.
    - **So our coordinate would be `x + 99` and `y + 99`.** In this case, it would be `(99, 99)`.

- Our first square would therefore be formed from `(0, 0)` to `(99, 99)`, and it would be 100px tall and wide.
- Lastly, we want the square to be white, so we use `fill` and set it to `color` which is `white`.

MILESTONE 1B: DRAW A ROW OF SQUARES

- We have code that can draw one square, so we should write a loop to loop it all the way through now.
- Recall that we have 8 columns (`N_COL = 8`) so we can write a `for each` loop that loops 8 times.

```
# loops num of Columns to draw 1 row of checkers
for col in range(N_COLS):
  draw_square(canvas, row, col)
```

- Variable `col` stands for columns, and range is the constant `N_COL` since we want it to loop for all our columns.
- Within the loop, it will call `draw_square()` and pass arguments `canvas` (from the Premilestone Phase), `col` for our current column iteration in the `for each` loop, and `row` for our current row iteration in the `nested for` loop.

Here's all of our code for Milestone 1 so far!

```
# creates the canvas
def main():
  canvas = make_canvas(CANVAS_SIZE, CANVAS_SIZE, 'Chess Board')
  # loops num of Columns to draw 1 row of checkers
  for col in range(N_COLS):
    draw_square(canvas, row, col)
  canvas.mainloop()

# draws one square
def draw_square(canvas, row, col):
  x = col * SIZE
  y = row * SIZE
  color = 'white'
  canvas.create_rectangle(x, y, x + SIZE, y + SIZE, fill=color)
```

```
Constants:
CANVAS_SIZE = 800

N_ROWS = 8
N_COLS = 8
SIZE = CANVAS_SIZE / N_ROWS - 1
```

Notes by Regina Gerbeaux

🎶meme break because this shit is hard 🎶

MILESTONE 2: DRAW A GRID OF SQUARES

- A **grid** is simply *multiple rows of squares*! So, we should make Milestone 1 loop for the number of rows we need!
- The best way to accomplish this is…doing a `nested for` loop with the `for each` loop we wrote above, since we want the code inside (`for each` loop) to repeat for every row.
- See below for our code. (The pre-written code has been greyed out for clarity in the new line of code we've added.)

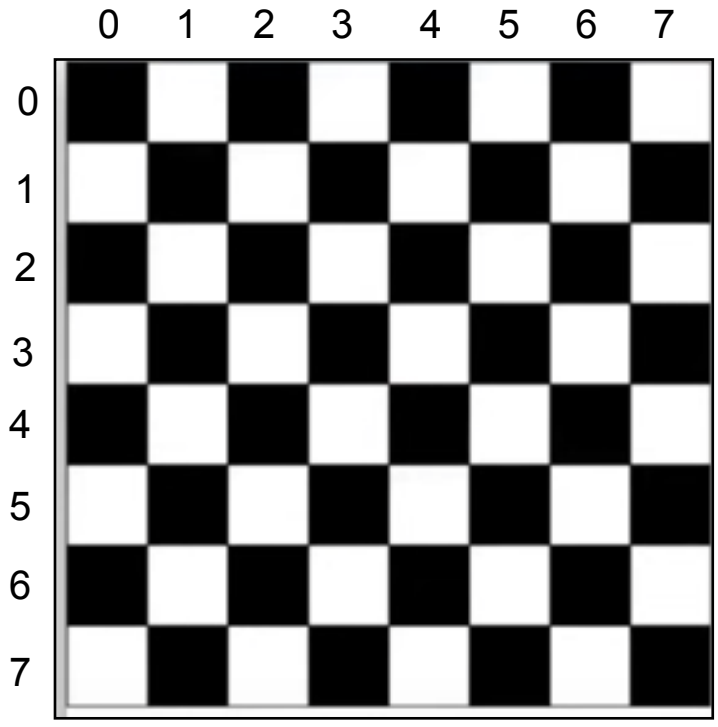
```
# our constants for canvas size, rows/columns, size of square
CANVAS_SIZE = 800

N_ROWS = 8
N_COLS = 8
SIZE = CANVAS_SIZE / N_ROWS - 1

# creates the canvas
def main():
  canvas = make_canvas(CANVAS_SIZE, CANVAS_SIZE, 'Chess Board')

  # loops num of Rows to draw multiple rows of checkers
  for row in range(N_ROWS):
    # loops num of Columns to draw 1 row of checkers
    for col in range(N_COLS):
      # draws single square
      draw_square(canvas, row, col)
  canvas.mainloop()

# draws one square
def draw_square(canvas, row, col):
  x = col * SIZE
  y = row * SIZE
  color = 'white'
  canvas.create_rectangle(x, y, x + SIZE, y + SIZE, fill=color)
```

MILESTONE 3: COLOR SQUARES TO FORM CHECKER PATTERN

- Recall above we hardcoded the color `'white'` in our `draw_square()` function (Milestone 1A) above:

```
# draws one square
def draw_square(canvas, row, col):
  x = col * SIZE
  y = row * SIZE
  color = 'white'
  canvas.create_rectangle(x, y, x + SIZE, y + SIZE, fill=color)
```

- But in reality, we want the color to alternate between black and white, as checkers are supposed to do!
- Let's redo it to get rid of the hard coded color, and make our code smart enough to alternate colors!
- A clean way to organize our code is to create a separate function responsible for coloring each of the squares either black or white.
- Let's declare a new function for our color, called `get_color()`.
  - Within the function, we'll write some logic to determine whether a square should be `black` or `white`.
  - Let's think about the squares' colors as they pertain to our columns (x values) and rows (y values).



- Notice that these are the squares that are supposed to be white: (1, 0), (0, 1), (3, 0), (2, 1), (1, 2), (0, 3), (5, 0), (4, 1), (3, 2), (2, 3), (1, 4), (0, 5)…etc.
- Notice these square are supposed to be black: (0, 0), (2, 0), (1, 1), (1, 2), (4, 0), (3, 1), (2, 2), (1, 3), (0, 4), ….etc.
- One thing we can notice from this pattern is **if we add up the value of the square's row and column**, white squares add up to an ODD number, and black squares add up to an EVEN number.
- Therefore, we can write some logic that says, "If the sum of my row and column is odd, color it white. Otherwise, color it black."

```
def get_color(row, col):
  if (row + col) % 2 == 0:
    return 'black'
  else:
    return 'white'
```

- Here's an example broken down in our function.
  - Say we have the coordinate (2, 1) which is supposed to be white. 2 + 1 = 3, and 3 is an odd number. Therefore, because 3 % 2 has a remainder, it returns white.
  - What about another coordinate, like (5, 5)? 5 + 5 = 10, and 10 is an even number. Therefore, because 10 % 2 == 0, it returns black.
- Lastly, we'll want to update `color` in `draw_square()` to remove the hard coded `white` and make it call our new `get_color()` function instead.
- Here's the updated code (pre-written code greyed out):

```
# draws one square
def draw_square(canvas, row, col):
  x = col * SIZE
  y = row * SIZE
  color = get_color(row, col)
  canvas.create_rectangle(x, y, x + SIZE, y + SIZE, fill=color)
```

Here is our final code!

```
# our constants for canvas size, rows/columns, size of square
CANVAS_SIZE = 800

N_ROWS = 8
N_COLS = 8
SIZE = CANVAS_SIZE / N_ROWS - 1

# creates the canvas
def main( ):
  canvas = make_canvas(CANVAS_SIZE, CANVAS_SIZE, 'Chess Board')

  # loops num of Rows to draw multiple rows of checkers
  for row in range(N_ROWS):
    # loops num of Columns to draw 1 row of checkers
    for col in range(N_COLS):
      draw_square(canvas, row, col)
  canvas.mainloop( )


# draws one square
def draw_square(canvas, row, col):
  x = col * SIZE
  y = row * SIZE
  color = get_color(row, col)
  canvas.create_rectangle(x, y, x + SIZE, y + SIZE, fill=color)


# determines the color of one square
def get_color(row, col):
  if (row + col) % 2 == 0:
    return 'black'
  else:
    return 'white'
```

Notes by Regina Gerbeaux

CS106A Notes - Python

Week 4 Notes: Animation (5/8/2020 Lecture)

Recap on 5/6/2020 Lecture

MAKING GRAPHICS

- To do something graphical, you have to use **tkinter**. `tkinter` is the Python graphics library.
  - To use `tkinter`, write the import statement `import tkinter`
  - This gives you access to all the graphical libraries
- You'll also need to **create a canvas** to put your graphic(s) on. To do this, CS106A provides the code to `make_canvas()` as such:

```
def make_canvas(width, height, title):
    ... [rest of code goes here]
```

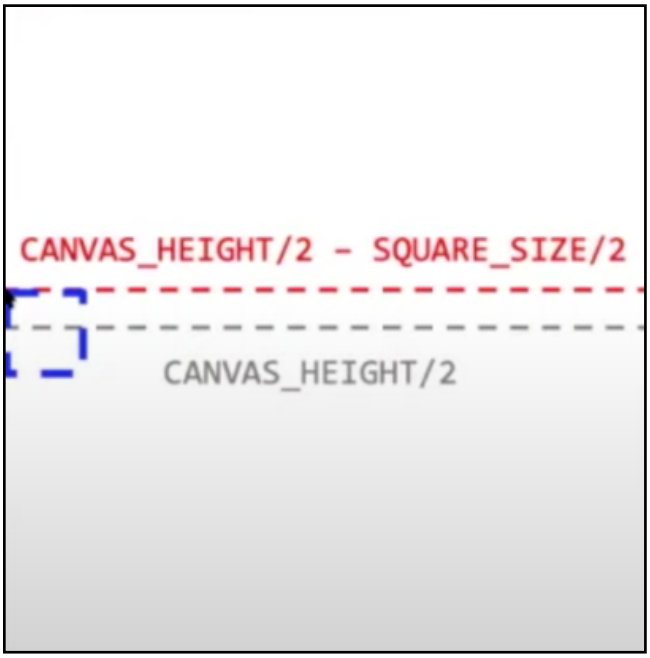
Because CS106A provides the code for us already, you don't have to write it yourself — but know that if you want to create your own, you'll need this piece of code.

MAKING A BLACK SQUARE

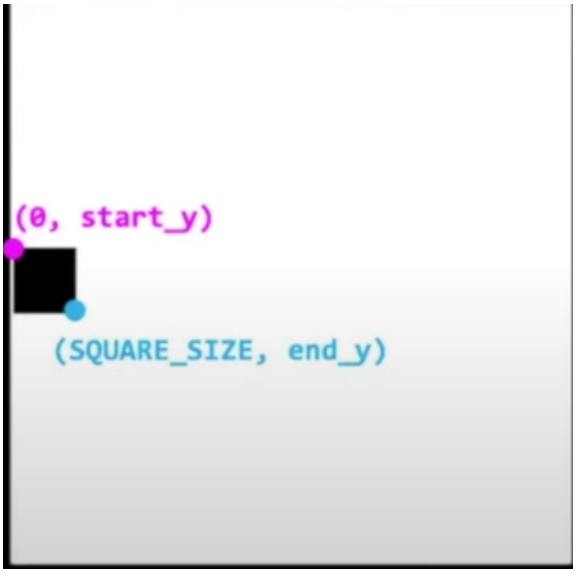
- The following are the steps to create a black square starting on the left side of the canvas, in the middle of the column.

```
def main():
  canvas = make_canvas(CANVAS_WIDTH, CANVAS_HEIGHT, 'Move Square')
  start_y = CANVAS_HEIGHT / 2 - SQUARE_SIZE / 2
  end_y = start_y + SQUARE_SIZE
  rect = canvas.create_rectangle(0, start_y, SQUARE_SIZE, end_y,
fill='black')
  canvas.mainloop()
```

- Call the `make_canvas()` function with arguments of:
  - canvas width (calling the constant `CANVAS_WIDTH`)
  - canvas height (calling the constant `CANVAS_HEIGHT`)
  - name of canvas (`'Move Square'`)
- Calculate location of square's top leftmost corner by:
  - using `CANVAS_HEIGHT / 2` (finds the middle of the canvas)
  - using `SQUARE_SIZE / 2` (finds the middle of the square) - square size is saved as a constant in the beginning
  - subtract the values from each other to **get the square directly in the middle of the canvas height**



- Calculate location of square's bottom rightmost corner by:
  - using calculated result of `start_y` and adding `SQUARE_SIZE` (specified as constant in the beginning)
- Create variable `rect` and call `create_rectangle()` on `canvas` with the following arguments:
  - the top leftmost corner coordinate, written as `(0, start_y)`
  - the bottom rightmost corner coordinate, written as `(SQUARE_SIZE, end_y)`
  - a fill argument (`fill='black'`)
- use `canvas.mainloop()` to keep the canvas displayed



Note: some "heavy duty" variables allow you to call functions on them — in this case, we call `create_rectangle()` *on* `canvas`.

Animation Loop

HOW TO ANIMATE MOVIES OR GAMES

- We'll learn how to move the square we drew during review to the middle of the screen. This is the animation version of `helloworld`!
- To animate something, we'll need to create an **animation loop**.
  - Animation works a lot like *frames* and *heartbeats*. It draws one frame, waits a heartbeat, draws the next frame, and so on. (For example, movies will contain about 50 frames per second.)
  - To our eyes, this creates a perception of motion!
- **Components of a program that animates something:**

```
def main():
    # setup

    while True:
        # update world

        # pause
        time.sleep(DELAY)
```

This is where we make all the variables we need and set our code up to animate!

Animation loop runs until finished.

Step 1 - Update World

Step 2 - Implement delay

  - **Setup code** - where we write our variables and set up whatever code we need to animate
  - **Animation loop** - the animation loop runs until it's finished (or, in some cases, it never ends.) Within the loop:
    - Step 1: Update the world forward one frame with each heartbeat
    - Step 2: Wait (pause) - if you don't pause, humans won't be able to see it.

Problem Solving: `move_rect.py`

MOVE THE SQUARE!

```
def main():
  canvas = make_canvas(CANVAS_WIDTH, CANVAS_HEIGHT, 'Move Square')
  start_y = CANVAS_HEIGHT / 2 - SQUARE_SIZE / 2
  end_y = start_y + SQUARE_SIZE
  rect = canvas.create_rectangle(0, start_y, SQUARE_SIZE, end_y, fill='black')
```

Part 1: "Setup Code"
Here, we've created our shape and stored it into a variable called rect.

- When we create a rectangle in `main()`, the value in `rect` is *returned* to us.
- When we create shapes, they are returned to us so **we need to save them into a variable**. That way, we can call them in our animation loop later on!

```
#within main()

  while True:
    # update world
    canvas.move(rect, 1, 0)
    canvas.update()

    # pause
    time.sleep(1/50)
```

Part 2: Animation Loop - update the world & wait
We set up a loop that will update the frame and wait

- Now that our variables are made, we can write our animation loop.
  - The first part will move our square using `.move()` from `tkinter` on `canvas`.
    - We pass into `.move()` what we want it to move (`rect`)
    - We pass in how much we want it to move to the right (`1`) — as a note, if you want to move to the left, give it a negative value.

    - We pass in how much we want it to move down (`0`) — as a note, if you want to move to the top, give it a negative value.
- Next, we tell `canvas` to `update` after we're done with the updates, to tell the program to draw it to the screen
- The second part will pause our loop for a specified amount of time.
    - In our example, it pauses it for `1/50th` of a second. This is our `DELAY` time.
    - We can do this by using the `.sleep()` function from the `time` library we `import` in the beginning.

MOVE IT TO CENTER

- So far, we've written the code to make a black square and move it.
- However, when we run the program, the square ends up running off the screen! *What?!*

- The reason the square won't stop is because we put our logic in a `while True` loop. Thus, the square continues moving indefinitely off our canvas.
- Recall **we want the square to stop at the midpoint of our canvas.** Therefore, we can write logic that will run our `loop` as long as the square is to the *left* of the midpoint in our canvas.
- In order to do this, **we'll need to get the x location of the square.**
  - In this course, Chris has provided us with helper methods (because we haven't learned about *lists* yet — the stuff in [brackets]. We'll learn about this next lecture!)
  - For now, these two helper functions will give us the x and y coordinates, respectively.

```
def get_left_x(canvas, object):
  return canvas.coords(object)[0]

def get_top_y(canvas, object):
  return canvas.coords(object)[1]
```

Note - we're not using `get_top_y()` *since we're simply moving the square horizontally, but the function is provided in case you wanted to know what that looks like.*

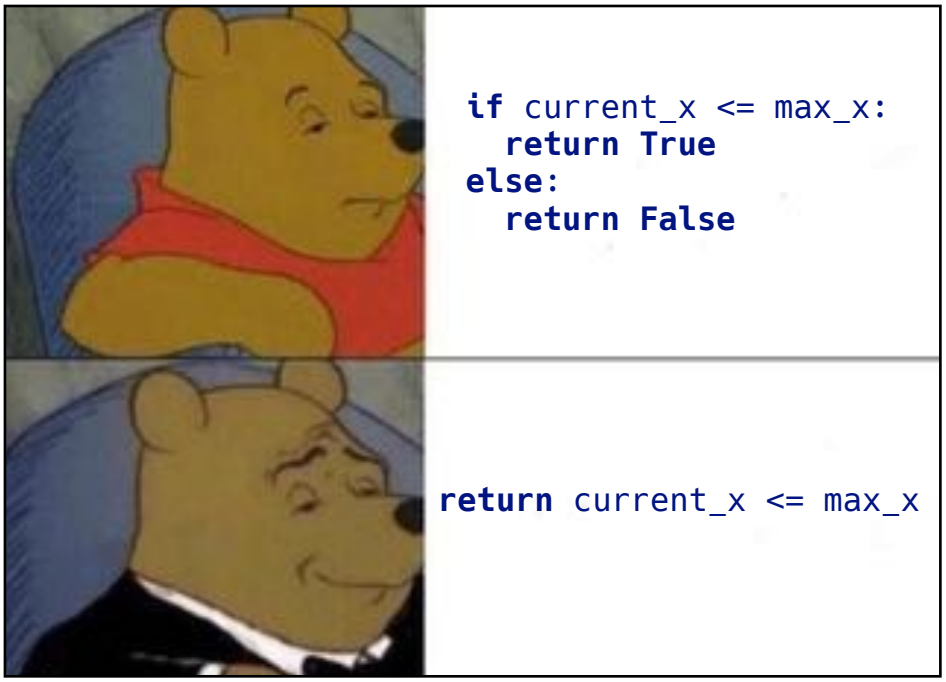
- Now that we know how to get the coordinate of our square, we can write logic to **replace** our `while True` logic.
- To make it clean, we'll use a helper function:

```
def is_rect_left_of_center(canvas, rect):
  current_x = get_left_x(canvas, rect)
  max_x = (CANVAS_WIDTH / 2) - (SQUARE_SIZE / 2)
  return current_x <= max_x
```

  - In our new function, `is_rect_left_of_center()`, we will check if the square is left of the center or not.
  - We call the helper function `get_left_x` and get the current location of our x coordinate. We save this to `current_x`.
  - We set `max_x` to `(CANVAS_WIDTH/2)-(SQUARE_SIZE/2)`, because `CANVAS_WIDTH/2` marks the midpoint of the canvas,

and `SQUARE_SIZE/2` marks the midpoint of the square. (We want the middle of the square to line up with the middle of the canvas.)

- As long as `current_x` is less than `max_x`, our square is still not at the middle; thus, it should continue moving. Thus, writing `return current_x <= max_x` will return a `True` or `False` value to our `while` loop.



- Back to `main()`, we can update the `while True` logic to call `is_rect_left_of_center()`.

```
#within main()

  ~~while True:~~
  while is_rect_left_of_center(canvas, rect):

    # update world
    canvas.move(rect, 1, 0)
    canvas.update()

    # pause
    time.sleep(1/50)
```

Notes by Regina Gerbeaux

Problem Solving: `ball.py`

BOUNCING BALL

- This problem will help us create a ball that moves until it hits a wall, and changes directions. We'll go through each frame together. :)

```
CANVAS_WIDTH = 600
CANVAS_HEIGHT = 600
CHANGE_X_START = 10
CHANGE_Y_START = 7
BALL_SIZE = 70
```

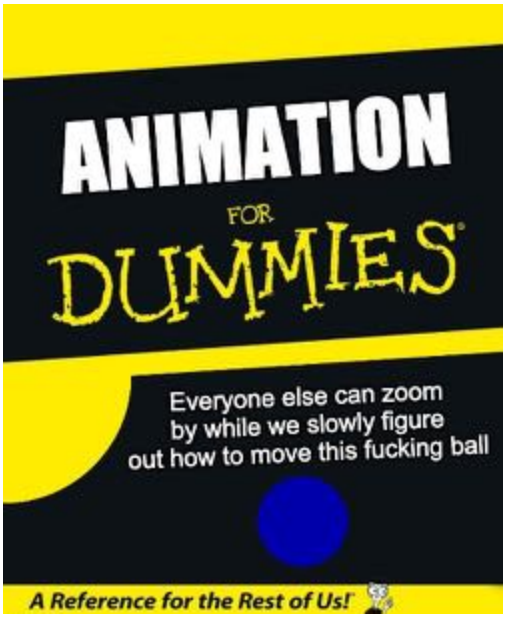
```
def main():
    canvas = make_canvas(CANVAS_WIDTH, CANVAS_HEIGHT, 'Bouncing Ball')

    ball = canvas.create_oval(0, 0, BALL_SIZE, BALL_SIZE, fill='blue', outline='blue')

    dx = CHANGE_X_START
    dy = CHANGE_Y_START
```

```
    while True:
        # update world
        canvas.move(ball, dx, dy)
        if hit_left_wall(canvas, ball) or hit_right_wall(canvas, ball):
            dx *= -1
        if hit_top_wall(canvas, ball) or hit_bottom_wall(canvas, ball):
            dy *= -1
        # redraw canvas
        canvas.update()
        # pause
        time.sleep(1/50)
```

```
def hit_left_wall(canvas, object):
    return get_left_x(canvas, object) <= 0

def hit_top_wall(canvas, object):
    return get_top_y(canvas, object) <= 0

def hit_right_wall(canvas, object):
    return get_right_x(canvas, object) >= CANVAS_WIDTH

def hit_bottom_wall(canvas, object):
    return get_bottom_y(canvas, object) >= CANVAS_HEIGHT
```

```
######## These helper methods use "lists" ###########
### Which is a concept you will learn Monday ###########

def get_left_x(canvas, object):
    return canvas.coords(object)[0]

def get_top_y(canvas, object):
    return canvas.coords(object)[1]

def get_right_x(canvas, object):
    return canvas.coords(object)[2]

def get_bottom_y(canvas, object):
    return canvas.coords(object)[3]
```

Part One: Constants



- We have some constants defined in the beginning to help us with the rest of our code:

```
CANVAS_WIDTH = 600
CANVAS_HEIGHT = 600
CHANGE_X_START = 10
CHANGE_Y_START = 7
BALL_SIZE = 70
```

- `CANVAS_WIDTH` and `CANVAS_HEIGHT` will help us set the dimensions of canvas. In this case, canvas will be `600px` by `600px`.
- `CANVAS_X_START` and `CANVAS_Y_START` will help us set how far our ball moves *right and down* or *left and up* depending on which direction the ball is facing. In this case, our ball will move `10px` down and `7px` right when facing east, or `10px` up and `7px` left when facing west — (when we switch directions — *we'll do this in our animation loop later*.)
- Lastly, `BALL_SIZE` will help us set our ball size. In this case, the program will fit the largest possible oval into a `70` by `70` rectangle.

Part Two: Setup

```
def main():
  canvas = make_canvas(CANVAS_WIDTH, CANVAS_HEIGHT,
                       'Bouncing Ball')
  ball = canvas.create_oval(0, 0, BALL_SIZE, BALL_SIZE,
                            fill='blue', outline='blue')
  dx = CHANGE_X_START
  dy = CHANGE_Y_START
```

- Recall from earlier before we jump into our animation loop, we have to *set up our code*. This is the part where we declare the variables we'll be using in our loop.
- First, we create `canvas` and set its value to `make_canvas()` provided to us in CS106A.
  - There, we pass in parameters of the `canvas` `WIDTH` and `HEIGHT` as specified in our constants.
  - We name `canvas` "Bouncing Ball."
- Next, we create `ball` and set its value to the `canvas` function `.create_oval()` (*as a reminder, it will fit the largest possible oval into the rectangle made of the dimensions specified.*)
  - Pass in parameters `(0, 0)` as the location of the top left corner, and (`BALL_SIZE`, `BALL_SIZE`) as the location for the bottom right corner.
  - The circle generated will be the *largest possible circle* that fits in between the points `(0, 0)` and (`BALL_SIZE`, `BALL_SIZE`).
  - Fill it blue by using `fill='blue'`.
  - Outline it blue by using `outline='blue'`.
- Lastly, set variables `dx` and `dy` to have values `CHANGE_X_START` and `CHANGE_Y_START` respectively.
  - `dx` stands for "Direction X" and `dy` stands for "Direction Y".
  - These will help us change the ball's direction when it bounces.

Part Three: Animation Loop

```
while True:
    # update world
    canvas.move(ball, dx, dy)
    if hit_left_wall(canvas, ball) or
        hit_right_wall(canvas, ball):
      dx *= -1
    if hit_top_wall(canvas, ball) or
        hit_bottom_wall(canvas, ball):
      dy *= -1
    # redraw canvas
    canvas.update()
    # pause
    time.sleep(1/50)
```

- We write our animation loop using **`while True`** because we want the ball to bounce forever.
- Within **update world:**
  - Call `.move()` from the `tkinter` library on `canvas` and pass in:
    - `ball` to move the ball (created in Part Two)
    - `dx` to move the ball to the right by `10px`
    - `dy` to move the ball down by `7px`
  - **`if`** the ball hits the left or right sides of the wall (checked by `hit_left_wall` and `hit_right_wall`), the `dx` value will multiply by `-1` to change the direction of the ball to move *left* (if it hits the right wall) or to move *right* (if it hits the left wall)
    - Remember that to move right, we use positive integers. To move left, we use negative integers.

    - Therefore, if it hits any side of the wall, we multiply it by `-1` to change the sign from either positive to negative, or negative to positive.
  - **`if`** the ball hits the top or bottom of the wall (checked by `hit_top_wall` and `hit_bottom_wall`), the `dy` value will multiply by `-1` to change the direction of the ball to move *up* (if it hits the bottom wall) or to move *down* (if it hit the top wall)
    - Remember that to move down, we use positive integers. To move up, we use negative integers.
    - Therefore, if it hits any side of the wall, we multiply it by `-1` to change the sign from either positive to negative, or negative to positive.
- To **redraw our canvas**, we'll update it using `canvas.update()`.
- To **pause**, use `time.sleep()` and pass in `1/50` of a second.

Part Four: Helper Functions

```
def hit_left_wall(canvas, object):
    return get_left_x(canvas, object) <= 0

def hit_top_wall(canvas, object):
    return get_top_y(canvas, object) <= 0

def hit_right_wall(canvas, object):
    return get_right_x(canvas, object) >= CANVAS_WIDTH

def hit_bottom_wall(canvas, object):
    return get_bottom_y(canvas, object) >= CANVAS_HEIGHT
```

- `hit_left_wall()` will determine if the ball has hit the left wall.
  - It accepts two arguments: `canvas` and `object`.
  - It calls `get_left_x()` *(which we'll go over below)* which returns the x coordinate to the left of the ball.
  - `hit_left_wall()` then sees if the returned x coordinate value is less than or equal to `0`.
  - If it's less than or equal to `0`, this means the object has hit the wall Therefore, the **`return`** value will be **`True`.**
    - (This is because the left wall has an `x` value of `0`.)
  - If it's not less than or equal to `0`, this means the object has not hit the wall. Therefore, the **`return`** value will be **`False`**.

- `hit_top_wall()` will determine if the ball has hit the top wall.
  - It accepts two arguments: `canvas` and `object`.
  - It calls `get_top_y()` *(which we'll go over below)* which returns the y coordinate at the top of the ball.

  - `hit_top_wall()` then sees if the returned y coordinate value is less than or equal to `0`.
  - If it's less than or equal to `0`, this means the object has hit the wall. Therefore, the **`return`** value will be **`True`.**
    - (This is because the top wall has a y value of `0`.)
  - If it's not less than or equal to `0`, this means the object has not hit the wall. Therefore, the **`return`** value will be **`False`**.

- `hit_right_wall()` will determine if the ball has hit the right wall.
  - It accepts two arguments: `canvas` and `object`.
  - It calls `get_right_x()` *(which we'll go over below)* which returns the x coordinate of the rightmost side of the ball.
  - `hit_right_wall()` then sees if the returned x coordinate value is greater than or equal to `CANVAS_WIDTH`.
  - If it's greater than or equal to `CANVAS_WIDTH`, this means the object has hit the wall. Therefore, the **`return`** value will be **`True`.**
    - (This is because the right wall has an `x` value of `CANVAS_WIDTH`.)
  - If it's not greater than or equal to `CANVAS_WIDTH`, this means the object has not hit the wall. Therefore, the **`return`** value will be **`False`**.

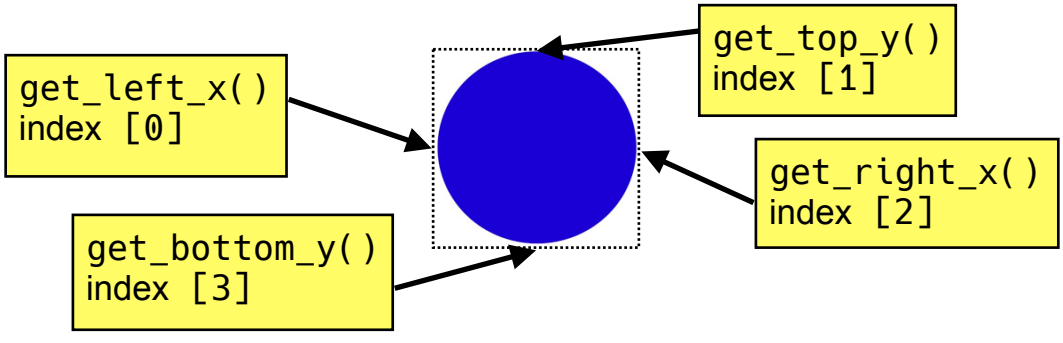
- `hit_bottom_wall()` will determine if the ball has hit the bottom of the wall.
  - It accepts two arguments: `canvas` and `object`.
  - It calls `get_bottom_y()` *(which we'll go over below)* which returns the y coordinate at the bottom side of the ball.
  - `hit_bottom_wall()` then sees if the returned y coordinate value is greater than or equal to `CANVAS_HEIGHT`.

- If it's greater than or equal to `CANVAS_HEIGHT`, this means the object has hit the wall. Therefore, the **`return`** value will be **`True`.**
  - (This is because the right wall has an `y` value of `CANVAS_HEIGHT`.)
- If it's not greater than or equal to `CANVAS_WIDTH`, this means the object has not hit the wall. Therefore, the **`return`** value will be **`False`**.

Part Five: Helper Method Using Lists

```
######## These helper methods use "lists" ###########
def get_left_x(canvas, object):
    return canvas.coords(object)[0]

def get_top_y(canvas, object):
    return canvas.coords(object)[1]

def get_right_x(canvas, object):
    return canvas.coords(object)[2]

def get_bottom_y(canvas, object):
    return canvas.coords(object)[3]
```

- These are helper methods that use something called **lists**. Lists are collections of items. We can access items in that list by referencing the *index number*. (But we'll learn more of this on Monday. :))
- All of these helper methods accept two arguments: `canvas` and `object`.
- `get_left_x()` will **`return`** the coordinates of `object` at index `[0]`, which is the x coordinate of the ball on the left side.
- `get_top_y()` will **`return`** the coordinates of `object` at index `[1]`, which is the y coordinate of the ball on the top side.
- `get_right_x()` will **`return`** the coordinates of `object` at index `[2]`, which is the x coordinate of the ball on the right side.
- `get_bottom_y()` will **`return`** the coordinates of `object` at index `[3]`, which is the y coordinate of the ball on the bottom side.



References

OBJECTS AND URLS (WAIT WHAT?)

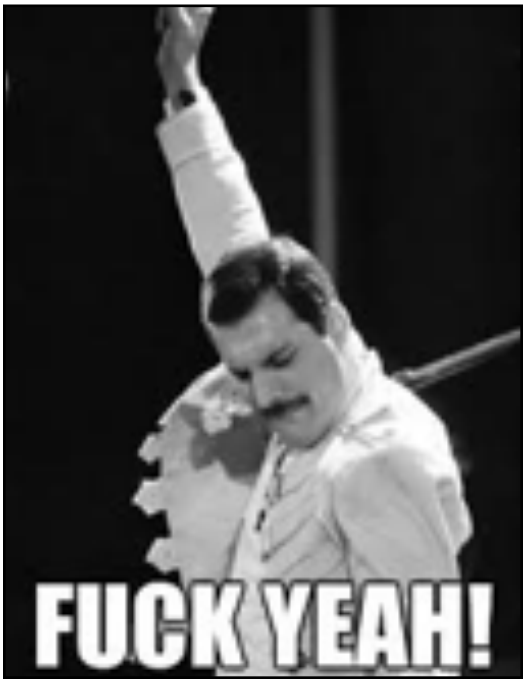
- Recall if you pass a parameter, what you get is a ***copy***.
- **Large variables (objects)** like *canvases* are stored using a reference. This reference gets copied when you pass the variable.
- This is similar to URLs, which *don't store information directly*, but store a reference to that information.

- Take a Google Doc for example: Let's say you're trying to share a Google Doc with your friend Chad (why do you have a friend named Chad? Never mind. Also, I'm very sorry if your name is Chad.)
- When you write a document using Google Docs, you can share it a few different ways.

    - You can copy and paste the entire thing in an email. But that doesn't sound like a smart way to do it: you'll be copying and pasting a *lot* of information, and you're passing a copy of that text to Chad.
    - Chad might want to make some edits to your Doc, and you'll want to see what those edits are. If Chad makes edits on his copy of

the document, it'll only show up on his copy and you won't get to see any of the edits!

- You can send a URL (link) of the Google Doc to Chad! Ahh. Now, Chad can edit the document directly, and that document will update with the changes on your end. You can see what valuable contributions Chad has made to the document.



- When you call `canvas = make_canvas()`, the program has a place in its memory called the **heap** where it stores really big things.
  - When we call `make_canvas()` and it makes the canvas, **the way it stores the canvas is similar to a URL**: the variable `canvas` refers to a *location in the **heap*** where your canvas lives (the URL of sorts.)
  - The heap is where the `make_canvas()` creation magic happens.
  - `canvas` stores the created canvas as its value.
- What if we wanted to create the blue ball from the previous example, and *tried saving the creation in a function?*

```
def main():
  canvas = make_canvas(…)
  make_ball(canvas)

def make_ball(canvas):
  canvas.create_oval(…, fill='blue')
```

doo, doodoo, doodoo, make canvas here

make a ball using a helper function

make ball in this function…?

- Recall from previous lectures that once a helper function runs, it wipes all local variables and memory. Therefore, when `make_ball()` is called, it also links to the heap, but doesn't save the oval that is created as nothing has been returned.
  - However, note that it doesn't destroy the canvas *itself*; it only destroys the link to the heap.
  - Because the heap works like a URL, it has now been changed by `make_ball()`! This is because **any changes made to the content of heap update on *all* ends**, like a Google Doc.

- Therefore, you could write code like this to use a helper function:

```
def main( ):
  canvas = make_canvas(…)
  ball = make_ball(canvas)

def make_ball(canvas):
  return canvas.create_oval(…, fill='blue')
```

- When passing variables, some variables act just like you are passing a URL. **That allows functions to <u>modify</u> the variable.**
- Other times, some variables act like you are passing the value. **This does not allow functions to modify the variable.**
- There is a difference between <u>copied values</u> versus <u>copied references to heaps</u>. Here is a list of variables and how they act:

Values are copied	**URLs** are copied
`boolean`	`canvas`
`integer`	`pixel`
`float`	`SimpleImage`
`string`	`list`

Notes by Regina Gerbeaux

Canvas Cheat Sheet

There are some more cool functions you can use to make your animations for your final project. Check these out!

MAKE A CANVAS

This code is usually provided by CS106A to create a canvas. If you want to create your own canvas and assign it to a variable in your final project, make sure to include this function in your code.

```
def make_canvas(width, height, title):

    top = tkinter.Tk()
    top.minsize(width=width, height=height)
    top.title(title)
    canvas = tkinter.Canvas(top, width=width + 1, height=height + 1)
    canvas.pack()
    return canvas
```

CREATE A SHAPE

Lets you create a shape and assign its value to a variable. Parts to replace with your own code have been italicized and marked in red.

```
name_of_variable = canvas.create_shape(starting_x,
              starting_y, ending_x, ending_y,
              fill=color_goes_here, outline=color_goes_here)
```

Example:

```
# creates a teal rectangle with a green border
# top left corner at (0, 0)
# bottom right corner at (10, 10).
rect = canvas.create_rectangle(0, 0, 10, 10,
          fill=teal, outline=green)
```

UPDATE CANVAS

After your changes have been made, update your canvas.

```
canvas.update( )
```

GET X LOCATION OF THE MOUSE

Find out where the mouse/pointer is located!

```
mouse_x = canvas.winfo_pointerx( )
```

MOVE A SHAPE TO A NEW COORDINATE

Lets you move a shape to a new location on your canvas.

```
canvas.moveto(shape_you_want_to_move, new_x_location,
              new_y_location)
```

Example:

```
# moves rectangle to (20, 20) on canvas
canvas.moveto(rect, 20, 20)
```

MOVE A SHAPE BY DETERMINED SLOPE

Lets you move a shape by specifying how much you want your shape to move by x and how much you want your shape to move by y.

```
canvas.move(shape_you_want_to_move, move_x_by_num_pixels,
            move_y_by_num_pixels)
```

Example:

```
# moves rectangle right by 5px and down by 10px
canvas.move(rect, 5, 10)
```

Another Example:

```
# moves rectangle left by 7px and up by 2px
canvas.move(rect, -7, -2)
```

RETRIEVE COORDINATES OF A SHAPE

Use if you want to know the location of your shape.

```
name_of_list = canvas.coords(shape)
```

Example:

```
# retrieves coordinate of rectangle
top_of_rect = canvas.coords(rect)
```

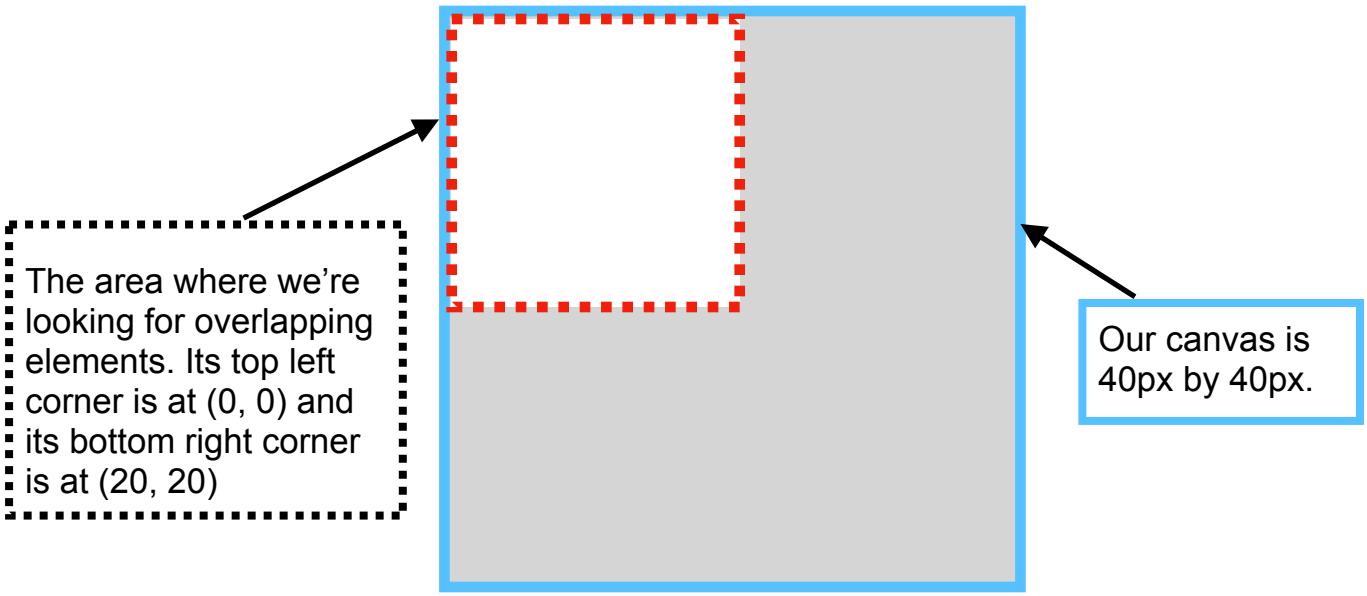
RETURN LIST OF ELEMENTS IN A RECTANGLE INTO LIST

Returns a list of elements within a given area.

```
result_variable = canvas.find_overlapping(x1, y1, x2, y2)
```

Example:

```
# returns a List of all the elements within the space from
    (0, 0) to (20, 20).
# assume our canvas is 40px by 40px.

results = canvas.find_overlapping
    (0, 0, 20, 20)
```



CHANGE COLOR OF SHAPE

Use to change the color of your shape.

```
canvas.itemconfig(shape, fill=new_color, outline=...)
```

Example:

```
# change the rectangle to fill red and outline yellow.

canvas.itemconfig(rect, fill=red, outline=yellow)
```

KEEP IMAGE ON SCREEN

Keep your canvas on display. You need this line in your code!

```
canvas.mainloop()
```

Notes by Regina Gerbeaux

CS106A Notes - Python
Week 5 Notes: Lists (5/11/2020 Lecture)

Introduction to Python Console

- Python is an **interpreted language**, which means it can run interactively using a "console".
- You can access the console by selecting "Python Console" in your PyCharm application.
- Within your console, you can type and execute Python statements and get instant results. The console prompts you by showing `>>>` to receive input:

The Console prompt

```
>>> x = 5
>>> x
5
```

Your inputs

The console's output

- We can perform more complex Python code in the console, too:

Console prompt changes to … to show it's in a loop.

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

hit backspace to exit indentation and exit the for loop

Prints result!

- The console is an easy place to test out how stuff works to answer questions you might have, and to see results quickly.
- Use `exit()` to leave the console.

And Then There Were None

- The term `None` refers to the notion of **no value**. If a variable is created *that doesn't return anything*, its value is `None`.

x has no value returned; it just prints "hi"

```
>>> x = print("hi")
>>> print(x)
None
```

So if we try to print the value of x, x has no value

Therefore, "None" prints because x holds no value

- If you compare anything with the value `None`, it evaluates to the Boolean `False`. (The exception, of course, is if you compare `None` to `None`, which is `True`.)

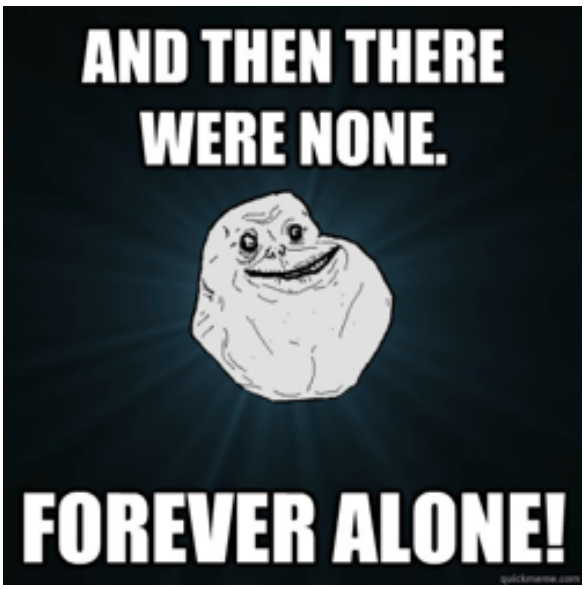
```
a = 'memes'
if a == None:
   ...
```

This evaluates to `False`, because `a` has a value.

```
a = None
if a == None:
   ...
```

This evaluates to `True`, because `None == None`

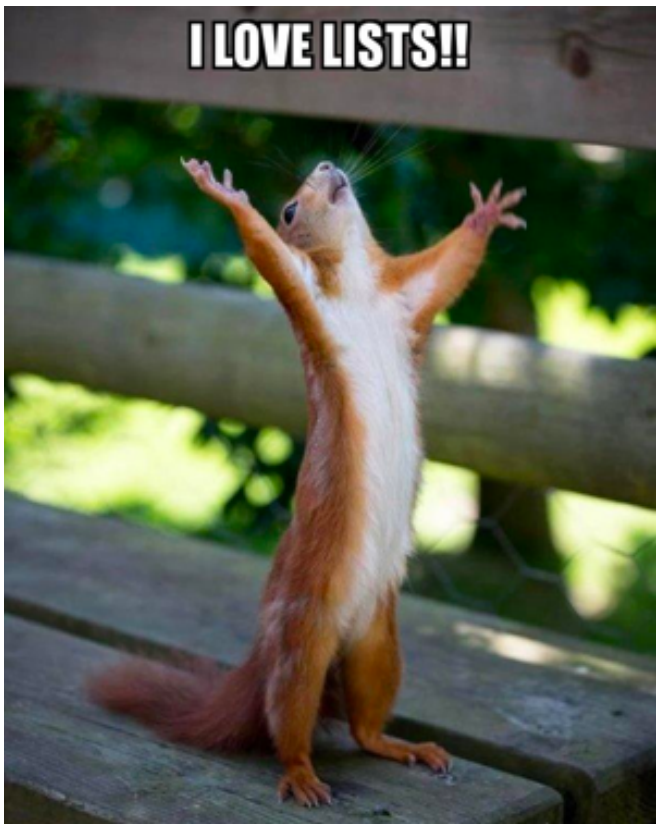
- `None` exists because sometimes we need to indicate our variable has *no* value.
- We'll find that being able to reference when a variable has *no* value will come in handy later on.



Notes by Regina Gerbeaux

Lists

INTRODUCTION TO LISTS



- A **list** is a way to keep track of an *ordered collection* of items.
  - The “items” are called **elements**.
  - “Ordered” refers to all the elements having a **position**.
  - A list can contain **multiple items**, which makes up a **collection.**
- Lists can *dynamically adjust its size* as elements are added or removed.
- You can add and remove stuff, which will automatically update the size of the list.
- Lists have lots of built-in functions that makes using them more straightforward.

MAKE A LIST

- To create a list, start and end with **[ ]** brackets.
- Separate elements with commas.

Brackets

List name

```
my_list = [1, 2, 3]
```

List elements separated by commas

This is a good time to plug this video for procrastination break: https://www.youtube.com/watch?v=6AXPnH0C9UA

- You can also make lists containing floats, strings, **images**, and even a mix of different value types:

```
reals = [4.7, -6.0, 0.22]
strs = [‘but’, ‘why’, ‘so’, ‘serious’]
mix = [‘whoa’, 2.3, ’we’, ‘can’, 37, ‘mix’, ‘value’, ‘types’, ‘?’, True]
```

- You can have an **empty list**. It’s a list with no elements in it.
- If you create an empty list, you are able to write code that can add elements to the list later on.

```
empty_list = []
```

For the reference: https://www.youtube.com/watch?v=1k8craCGpgs

- A **list with one element is not the same** as the **element** itself.

```
list_one = [1]
```

*This is a **list** with one element in it.*

```
one = 1
```

*This is a **variable** with a value of 1.*

```
list_one != one
```

These two are not the same, because [1] is not the same as 1. You have to pull the individual value out.

ACCESS AND UPDATE ELEMENTS IN LIST

- Lists are *ordered*, so you can access them by referencing their **index**. Indexes start from zero.
- To access an element, use the following format:

```
access_element = name_of_list[index_num_goes_here]
```

- For example, this is a list named `letters` that has five elements.

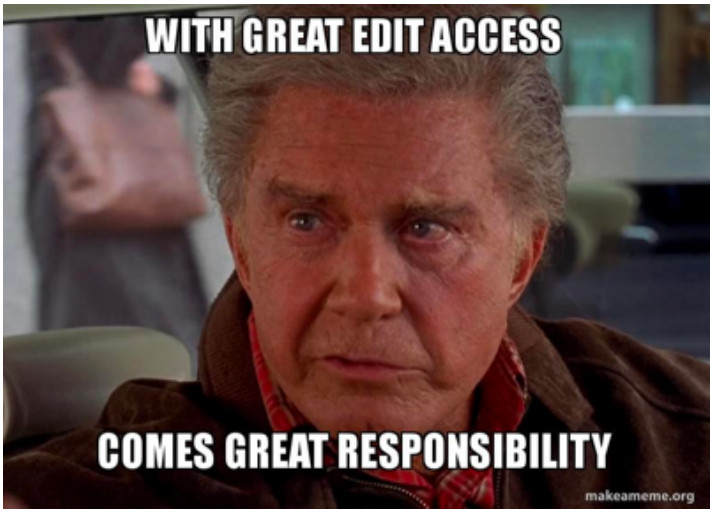
```
letters = ['a', 'b', 'c', 'd', 'e']
```

  - `'a'` is at **index** `0` because it is the **first element** in the list
  - `'b'` is at **index** `1` because it is the **second element** in the list
  - `'c'` is at **index** `2`, and so on.
  - To access the letter `'a'`, type `letters[0]` to access the Element in Index 0 (the first element in the list.) *You pronounce this as "letters subzero" referring to subscript, or element, zero.*
  - `letters[1]` is `'b'`, and so on.
- You can also **update** individual elements, just like regular variables:

```
letters = ['a', 'b', 'c', 'd', 'e']
letters[0] = 'x'
```

We updated `letters[0]` value to `'x'`, so now, `letters = ['x', 'b', 'c', 'd', 'e']`

- Note: this actually changes the element in the list.



GET LENGTH OF LIST

- We might want to **know how long our lists are**. To find list length, use the following format:

```
len(list_name)
```

- `len` stands for “length”, and entering this into our Console will return the number of items/elements in the list.

```
>>> letters = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
>>> len(letters)
5
```

- Note, the length of an empty list is still zero. It is *still* a list; it just has no elements in it.
- Since we can access the length of our list (which is a number), we can also use this in a loop:

`len(letters)` is 5, so our range will be 5.

```
>>> for i in range(len(letters)):
...         print(str(i) + “ -> ” + letters[i])
...
0 -> a
1 -> b
2 -> c
3 -> d
4 -> e
```

prints `i`, an arrow, and the element at `i`

our console output

- We can also **use negative indexing** to work from the end of our list. This is convenient when we want to quickly access the last element in our list (because then, we just use `-1`.)
- **To access an element from the end of our list**, use this format:

```
list_name[negative_num_goes_here]
```

  - For example:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> letters[-1]
e
>>> letters[-2]
d
```

This accesses the *last* element in the list

This accesses the *second to last* element in the list

- You can also think of a **negative index** as **the length of the list subtracted by that negative index**.

```
list[neg_index] = list[len(list) - neg_index]
```

  - For example, if `len(letters)` is 5, to access `'e'` (which is at Index `4`), we use negative index `-1`.

```
-1 = len(letters) - 1
# selects 'e'
```

Index -1 selects the same element as index 4.

  - Here's an easy way to visualize it:

```
letters[-1] = letters[len(letters) - 1]
letters[-1] = letters[5 - 1]
letters[-1] = letters[4]
```

  - We can also access `'c'` by using either Index `2` or Index `-3`.

```
-3 = len(letters) - 3
# selects 'c'
```

Index -3 selects the same element as index 2.

- Lastly, here is a comparison on negative versus positive indexes.

```
            -5   -4   -3   -2   -1
letters = ['a', 'b', 'c', 'd', 'e']
            0    1    2    3    4
```

Negative Indexing

Positive Indexing

- When we try to reference an index that is out of range, our Console will give us an **IndexError** bug.
- So don't get cute and type `letters[6]` when you know Index `6` is out of range, okay?

This baby is cute. Calling an index out of range is not.

BUILD YOUR LIST

- To **add elements to the end of the list**, use this format:

```
list_name.append(added_element)
```

- For example:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> letters.append('e')
>>> print(letters)
['a', 'b', 'c', 'd', 'e']
```

Add 'e' to letters

Console prints with added element

- You can also start with an **empty list** and use **`.append( )`** to build up your empty list:

```
>>> new_list = []
>>> new_list.append('element')
>>> print(new_list)
['element']
```

Added new element to list

Console prints with added element

REMOVE YOUR ELEMENTS

Because no one needs the air element…or whatever.

- `.pop( )` removes an element at end of list and returns the element.
- To **remove elements at the end of the list**, use this format:

```
list_name.pop(removed_element)
```

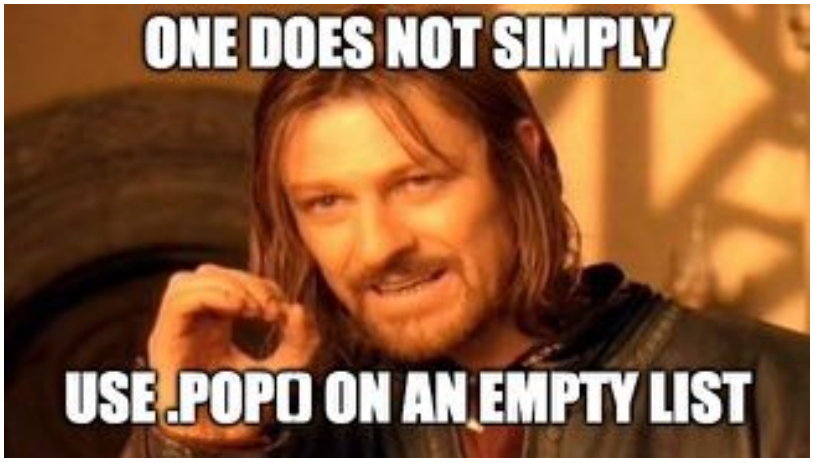
```
>>> avatar = [water, earth, fire, air]
>>> useless_element = avatar.pop( )
>>> useless_element
['air']
>>> useless_element = avatar.pop( )
>>> print(useless_element)
['fire']
```

Removes 'air' and returns it to the variable

Console prints the returned value

If you use .pop() again, it returns the new removed element

- When you try to use `.pop()` on an *empty list*, you'll also get an **IndexError** bug (just like when you try to reference a nonexistent element in a list!) So don't do it — it's not cute.



MOAR FUNZ WITH LISTZ

- To **print a list**, use this format:

```
print(name_of_list)
```

  - For example:

```
>>> num_list = [1, 2, 3, 4]
>>> print(num_list)
[1, 2, 3, 4]
```

- To **check if a list is empty**, use this format:

```
if name_of_list:
```

  - This expression evaluates to True or False. If a list has elements, it evaluates to True. Otherwise, an empty list evaluates to False.
  - For example:

```
if num_list:
  print('num_list is not empty')
else:
  print('num_list is empty')
```

- To **check if a list contains an element**, use this format:

```
if element in name_of_list:
```

  - This expression evaluates to True or False. If a list has that element, it evaluates to True. Otherwise, if the list doesn't contain that element, it evaluates to False.
  - For example:

```
good_guys = ['Leia', 'Luke', 'Han']

if 'Palpatine' in good_guys:
  print('You have a Villain in your list!')
else:
  print('You have a list of Rebels.')
```

  - You could use this in any kind of expression that evaluates to True or False.

LIST EXTRAVAGANZAAAA

REMOVE AND RETURN ELEMENT AT SPECIFIC INDEX

Remove a specific element by giving the index num as the parameter.

Use this format:

```
name_of_list.pop(index_number)
```

Example:

```
>>> fun_list = ['a', 'b', 'c']
>>> x = fun_list.pop(1)
>>> x
'b'
>>> fun_list
['a', 'c']
```

Removed element at Index 1, 'b', and returned it to x.

REMOVE AND RETURN FIRST OCCURRENCE OF ELEMENT

Remove the first occurrence of a specific element by giving the element's name as the parameter.

Use this format:

```
name_of_list.remove(name_of_element)
```

Example:

```
>>> outkast = ['hey', 'ya', 'hey', 'ya']
>>> x = outkast.remove('ya')
>>> outkast
['hey', 'hey', 'ya']
```

Removed first occurrence of 'ya' (located at Index 1) and returned it to x.

*Note: If you try to remove an element that isn't in the list, you'll get a **Value Error**.*

EXTEND A LIST WITH ANOTHER LIST

Adds all elements from one list to the list the function is called on.

Use this format:

```
name_of_list.extend(list_to_add_on)
```

Example:

```
>>> rebels = ['Luke', 'Leia', 'Han']
>>> droids = ['C3PO', 'R2D2']
>>> rebels.extend(droids)
>>> rebels
['Luke', 'Leia', 'Han', 'C3PO', 'R2D2']
```

Added elements from `droids` to `rebels` list.

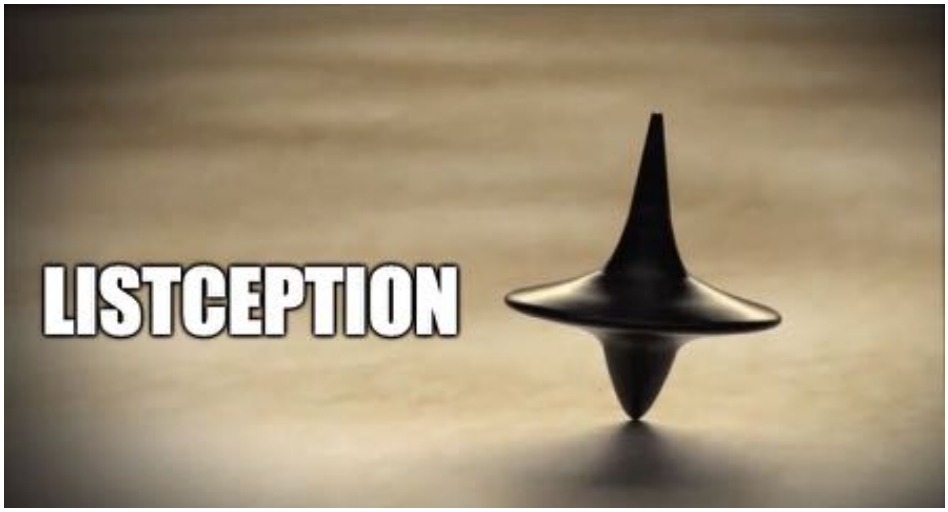
Note: **`.extend( ) doesn't alter the added list.`** If we printed `droids` into console, it would still list `['C3PO', 'R2D2']` as its elements.

Another Note: **append is not the same as extend!** Append **adds a *single element***, whereas extend ***merges one list into another***.

This is what it would look like if we used **`.append( )`** to the list

```
>>> rebels = ['Luke', 'Leia', 'Han']
>>> droids = ['C3PO', 'R2D2']
>>> rebels.append(droids)
>>> rebels
['Luke', 'Leia', 'Han', ['C3PO', 'R2D2']]
```

Notice `droids` was added as a list within the list.



ADD LISTS TOGETHER TO CREATE A NEW LIST

Adds elements together from two or more lists. Works just like `.extend( )` *except it creates a new list. Original lists unchanged.*

Use this format:

```
new_list_name = list_to_copy + other_list_to_copy
```

Example:

```
>>> og_char = ['Luke', 'Leia', 'Han']
>>> new_char = ['Rey', 'Poe', 'Finn']
>>> resistance = og_char + new_char
>>> resistance
['Luke', 'Leia', 'Han', 'Rey', 'Poe', 'Finn']
```

Make new list called `resistance` formed by the elements in `og_char` and `new_char`

ADD ELEMENTS FROM ONE LIST TO ANOTHER

Adds elements from one list to another list just like `.extend()`.

Use this format:

```
list_to_update += list_to_add_elements_from
```

Example:

```
>>> rebels = ['Luke', 'Leia', 'Han']
>>> droids = ['C3PO', 'R2D2']
>>> rebels += droids
>>> rebels
['Luke', 'Leia', 'Han', 'C3PO', 'R2D2']
```

Added elements from `droids` to `rebels` list.

FIND INDEX OF FIRST INSTANCE OF ELEMENT IN LIST

Searches for and returns the index of the first instance of the element.

Use this format:

```
name_of_list.index(element_name)
```

Example:

```
>>> rebels = ['Luke', 'Leia', 'Han']
>>> rebels.index('Leia')
1
```

Returns Index 1.

Searches for `'Leia'` from `rebels` list.

Note: if you try to search for an element that isn't on the list, you will get a ***ValueError****.*

ADD ELEMENT AT A SPECIFIC INDEX

Add an element at a given index. All other elements will shift down.

Use this format:

```
name_of_list.insert(index number, element_name)
```

Example:

```
>>> jedi = ['Luke', 'Obiwan']
>>> jedi.insert(1, 'Mehran')
>>> jedi
['Luke', 'Mehran', 'Obiwan']
```

Adds 'Mehran' to `jedi` list at Index 1.

MAKE A COPY OF A LIST

Makes a copy of one list and saves it to another list. This does not change the original list that has been copied.

Use this format:

```
new_list_name = name_of_list_copied.copy()
```

Example:

```
>>> actual_jedi = ['Luke', 'Obiwan']
>>> fantasy_list = actual_jedi.copy()
>>> fantasy
['Luke', 'Obiwan']
>>>fantasy.insert(1, 'Mehran')
>>> fantasy
['Luke', 'Mehran', 'Obiwan']
>>> actual_jedi
['Luke', 'Obiwan']
```

Creates new list called `fantasy_list`

We can make changes to `fantasy_list` without changing the original list

`actual_jedi` remains unchanged.

GET MAX VALUE IN A LIST

Get the largest number in a list.

```
list = [3.6, 2.9, 8.0, -3.2, 0.5]
```

Use this format:

```
max(name_of_list)
```

Example:

```
>>> max(list)
8.0
```

GET MIN VALUE IN A LIST

Get the smallest number in a list.

Use this format:

```
min(name_of_list)
```

Example:

```
>>> min(list)
-3.2
```

GET SUM OF VALUES IN A LIST

Get the sum of values in a list.

Use this format:

```
sum(name_of_list)
```

Example:

```
>>> sum(list)
11.8
```

FOR LOOP THROUGH A LIST'S ELEMENTS

Iterate through a list's elements using a `for` *loop.*

```
list = ['leia', 'luke', 'han']
```

Use this format:

Retrieves length of `list_name`

```
for i in range(len(list_name)):
  element = list_name[i]
  print(element)
```

Prints element at the current index

Example:

```
>>> for i in range(len(list)):
...     element = list[i]
...     print(element)
...
Leia
Luke
Han
```

FOR-EACH LOOP THROUGH A LIST'S ELEMENTS

You can accomplish the same thing above using a `for-each` *loop.*

Use this format:

```
for element in list_name:
  do something with element
```

Example:

```
>>> for element in list:
...     print(element)
...
Leia
Luke
Han
```

Note that lists are collections. Just like images are collections of pixels, lists are collections of elements.

Lists as Parameters

PASSING VARIABLES AS PARAMETERS

- Recall from our previous lecture that certain variables act like **copies** when passed as parameters and are therefore ***immutable***.
- These primitive types include:
  - integer
  - float
  - Boolean
  - string
- Other “more robust” or heavy variables act like **URLs** or references to memory (called *heaps*) when passed as parameters and are therefore ***mutable***.
- These include:
  - canvas
  - pixel
  - SimpleImage
  - **list** (now you know it is!)
- When you pass a **list** as a parameter, you pass a **reference** (the URL) to the *actual list, not a copy*. Therefore, you can make changes to the original.
  - This is called a **pass-by-reference** because you’re passing a reference to the location of the original itself, not a copy.
  - Therefore, in functions, **changes to values in the list <u>persist</u> after functions end** (unlike previous functions where objects remain unchanged unless <u>returned</u> as a value to the function caller.)

- For example:

```
>>> def add_five(num_list):
...     for i in range(len(num_list)):
...         num_list[i] += 5
…
>>> def main():
...     values = [5, 6, 7, 8]
...     add_five(values)
...     print(values)
...
[10, 11, 12, 13]
```

Will add 5 to every number in the list

Original list

Run add_five function on original list

numbers in original list are now updated

- However, **when you create a new list in a function, you are no longer dealing with the list that has been passed in as a parameter.** Any changes made to the new list will not show up on your original list that was passed in as the parameter.
- As soon as you create a new list (even with the same name), you are now pointing to a new URL and not referring to the original passed-in list.

```
>>> def create_new_list(num_list):
...     num_list.append(9)
...     num_list = [1, 2, 3]
...
>>> def main():
...     values = [5, 6, 7, 8]
...     create_new_list(values)
...     print(values)
...
[5, 6, 7, 8, 9]
```

Adds 9 to the current list

This creates a *new* list called num_list. It goes away once the function is done running.

Original list

Calls create_new_list()

numbers will print with just the appended num

LAST FEW NOTES ON LOOPS AND LISTS

```
list = [1, 2, 3]
```

- When using a `for` loop using `range`, you are **modifying the list in place:**

```
for i in range (len(list)):
  list[i] += 1  # modifying list in place.

# list is now [2, 3, 4]
```

- When using a `for-each` loop, you're **modifying the local variable of each value in the list. If the element is a primitive type, you're not changing the list!**

```
for element in list:
  element += 1

# modifying local variable of the element,
not the element in the list itself.
# list is still [1, 2, 3]
```

- **Therefore, if you want to make changes to the List itself and your List contains primitive types, you must use a `for` loop using `range` to reference the *list itself.***

Main Takeaways:

* Use **for loop with range** when *modifying primitive-type elements of list* (including ints, floats, and strings.)

* Use **for each loop** when *not modifying elements, or modifying non-primitive-type elements of list* (e.g.: pixels)

Problem Solving: averagescores.py

The purpose of this program is to get scores from user, add extra credit to their scores, and recompute the average.

```
EXTRA_CREDIT = 5

def get_scores( ):
   score_list = []
    while True:
        score = float(input("Enter score (0 to stop): "))
        if score == 0:
            break
        score_list.append(score)
    return score_list

def compute_average(score_list):
    """
    >>> compute_average([1.0, 2.0, 3.0, 4.0])
    2.5
    >>> compute_average([1.0, -1.0])
    0.0
    """
    num_scores = len(score_list)
    total = sum(score_list)
    return total / num_scores

def give_extra_credit(score_list, extra_credit_value):
    for i in range(len(score_list)):
        score_list[i] += extra_credit_value

def print_list(alist):
    for value in alist:
        print(value)

def main( ):
    scores = get_scores( )
    print("Scores are:")
    print_list(scores)
    avg = compute_average(scores)
    print('Average score is ' + str(avg))

    print('Adding extra credit: ' + str(EXTRA_CREDIT) + " points")
    give_extra_credit(scores, EXTRA_CREDIT)
    print("New scores are:")
    print_list(scores)
    avg = compute_average(scores)
    print('New average is ' + str(avg))
```

Extra Credit constant is 5

Ask user for list of scores (valid numbers.) Return list containing the scores.

Calculate average value of the list of scores. Doctests included.

Adds extra credit to all of the values in score_list.

Prints the values in the list passed in.

Computes average of test scores provided by user. Adds extra credit to scores, and recomputes the average. Prints new average.

Part 1: `get_scores( )`

- Create an empty list called `score_list`
- Within a `while` loop:
  - Receive user input for variable `score` and convert it to a float.
  - Keep running the loop until user enters `0`
  - If user input isn't `0`, **append** `score` to `score_list`
- Return the value of `score_list`

```
def get_scores( ):
   score_list = []
    while True:
        score = float(input("Enter score (0 to stop): "))
        if score == 0:
            break
        score_list.append(score)
    return score_list
```

Part 2: `compute_average( )`

- `compute_average( )` has a parameter of the list `score_list`
- Create variable called `num_scores` that has the value of `score_list`'s **`length`**.
- Create variable called `total` and set its value to the **`sum`** of the elements in `score_list`
- Return `total` divided by `num_scores` for the computed average
- **Doctests** to help us make sure our code works include:
  - `1.0 + 2.0 + 3.0 + 4.0 = (10.0) / 4 = 2.5`
  - `1.0 + -1.0 = (0.0) / 2 = 0.0`

```
def compute_average(score_list):
    """
    >>> compute_average([1.0, 2.0, 3.0, 4.0])
    2.5
    >>> compute_average([1.0, -1.0])
    0.0
    """
    num_scores = len(score_list)
    total = sum(score_list)
    return total / num_scores
```

Part 3: `give_extra_credit()`

- `give_extra_credit()` has two parameters: `score_list` (the list of scores) and `extra_credit_value` (the amount of extra credit to give.)
- Write a **`for`** loop that goes through all elements in `score_list`:
  - Add `extra_credit_value` to each element in the `score_list`.
- This updates all the values in our `score_list`.
- We use the **`for i in range`** because we want to *modify primitive types* (in this case, floats.)

```
def give_extra_credit(score_list, extra_credit_value):
    for i in range(len(score_list)):
        score_list[i] += extra_credit_value
```

Part 4: `print_list()`

- `print_list()` has a parameter called `alist` (a list of scores.)
- Write a **`for-each`** loop that goes through all elements in `alist`:
  - Print each element in the list (the values provided)
- We use the **`for-each`** loop because we are *printing and <u>not</u> modifying primitive types here.*

```
def print_list(alist):
    for value in alist:
        print(value)
```

Part 5A: Normal Scores in `main()`

- Create variable called `scores` and call `get_scores()`. `get_scores()` returns a **list** of `scores` as elements.
- `Print`, "**`Scores are:`**"
- Call `print_list()` with `scores` given as an argument. This will `print` all of the elements in the list.

- Create variable called `avg` and call `compute_average()` with `scores` given as an argument. This will **`return`** the `average` of the scores as `avg`'s value.
- `Print`, "**`Average score is`**" and concatenate with a stringified version of `avg`.

Part 5B: Extra Credit Scores in `main()`

- `Print`, "**`Adding extra credit:`**" and concatenate with a stringified version of the `EXTRA_CREDIT` constant and "**`points`**". (This prints, "**`Adding extra credit: 5 points`**")
- Call `give_extra_credit()` and pass in the list stored in `scores` as `score_list` followed by `EXTRA_CREDIT` as `extra_credit_value`.
- `Print`, "**`New scores are:`**"
- Call `print_list()` with the now-updated `scores` given as an argument. This will `print` all of the *newly updated elements* in the list.
- Update the value of `avg` to compute the *new average of* `scores` (now with the extra credit added) using `compute_average(scores)`
- Finally, `print`, "**`New average is`**' and concatenate with stringified version of the now-updated `avg`.

```
def main():
    scores = get_scores()
    print("Scores are:")
    print_list(scores)
    avg = compute_average(scores)
    print('Average score is ' + str(avg))

    print('Adding extra credit: ' + str(EXTRA_CREDIT) + " points")
    give_extra_credit(scores, EXTRA_CREDIT)
    print("New scores are:")
    print_list(scores)
    avg = compute_average(scores)
    print('New average is ' + str(avg))
```

CS106A Notes - Python
Week 5 Notes: Dictionaries (5/15/2020 Lecture)

Recap with Files

SYNTAX FOR ACCESSING FILES

- There are two ways to read files: `with` and `for` loops.
- **The first is using the `with` syntax:**
  - Creates an association with the `file` on disc (`mydata.txt` in the example) to `file` in Python
  - When we use `with open`, when Python is done reading the file, it automatically closes the file for us.
  - Reads the `file` on disc `line` by line using a `for` loop.
- We call `.strip()` on the lines to remove carriage returns (`\n`) to leave just the string that is the actual text of the line.

```
with open('mydata.txt') as file:
  for line in file:
    line = line.strip()
    print(line)
```

- The second way is using `for` loop syntax (the newer way to do this):
  - This works the exact same was as `with`, *except it doesn't automatically close the file for us.*
    - Note: This isn't a problem if you're not planning on doing anything else with the `file`.

```
for line in open('mydata.txt');
  line = line.strip()
  print(line)
```

REVIEW OF STRINGS

```
PUNCTUATION = '.!?,-:;'

def delete_punctuation(string):
result = ''
for char in string:
  if char not in PUNCTUATION:
    result += char
return result
```

- This function, `delete_punctuation()`, removes all punctuation marks and returns the resulting string, now punctuation-free.
- First, we define constant `PUNCTUATION` to determine what counts as a punctuation mark.
- Then, we create our function, `delete_punctuation()`, which accepts a `string` parameter.
- Create a variable called `result` and set its value to an empty string. This variable will hold our punctuation-free string.
- We write a **`for-each`** loop that iterates through every `character` within the passed-in `string`.
  - Within the loop, we write an if statement that checks if char is not a punctuation mark.
  - If `char` is not a punctuation mark, we **append** it to the rest of the non-punctuation characters stored in `result`.
- After our loop is done, we return our punctuation-free string stored in `result` to the function caller.
- This is what our console would look like if we tried this out. We pass in the `string 'REMOVE --the-- punctuation!!!!!'` and it **`returns`** the string-free `result`, `'REMOVE the punctuation'`.

```
>>> delete_punctuation('REMOVE --the-- punctuation!!!!!')
'REMOVE the punctuation'
```

READING LINES FROM A FILE

```
def count_words(filename):
  count = 0
  with open(filename, 'r'), as file:
    for line in file:
      line = line.strip()
      word_list = line.split()
      for word in word_list:
        print("#" + str(count) + ": " + word)
        count += 1
  print(filename + " contains " + str(count) + " words")
```

- This function, `count_words()`, takes a file and prints a statement that tells you how many words there are in the file. (Anything
- In `count_words()`, accept a `filename` parameter.
- Create the variable `count` and set its initial value to `0`.
- Open the `file` that has been passed in using the **`with open`** syntax. The `'r'` notation is old-school and means you're *passing in the file for Reading*. This step converts the filename into a `file` that Python can read and work with.
- Now that we've accessed `file` in a way Python can read, read each `line` and strip it of its carriage returns (`\n`) using `.strip()`
- Create a variable called `word_list` and set its value to be `line`'s value **split** between each space.
  - Recall that `.split()` will split a string into individual elements on every instance of the space.
- Then, write a **`for-each`** loop to iterate through every `word` within `word_list`. This loop will number each of the words from the file.
  - The loop will print a concatenation of "*`#count: word`*", and then increase the `count` by 1.
- Finally, after *all* of the lines have been iterated through, it will print a concatenation of "*`filename contains count words`*".

- Let's say this is the file text we're passing into our function:

```
# testfile.txt
Very few\n
words here.\n
```

- We pass `testfile.txt` as an argument to `filename`.
- Our program opens the file and begins iterating through every line in the file. The first `line` it will iterate through is "`Very few`".
  - The program ***strips*** the `line` of its carriage return (I've indicated the `\n` in light gray within the text so you understand it's there as a line break, but it's not actually visible in our text.)
  - It then turns it into List items stored in `word_list`, splitting at every instance of a space. This is what `word_list` looks like so far:

```
word_list ['Very', 'few']
```

  - We then iterate through every `word` in `word_list`, and **`print`** the number from `count` along with the `word`. Our `count` goes up by 1 each time. So far, this gets printed in our console.

```
#0: Very
#1: few
```

- We repeat this process to the next line. We'll iterate through "`words here.`", our next line.
  - Program ***strips*** the the `line` of its carriage return and turns it into List Items stored in `word_list`. This is now our `word_list`:

```
word_list ['Very', 'few', 'words', 'here']
```

- Then, we do the same thing and print the `count`, print the `word`, and increase the `count` by 1 each time. This is what's printed:

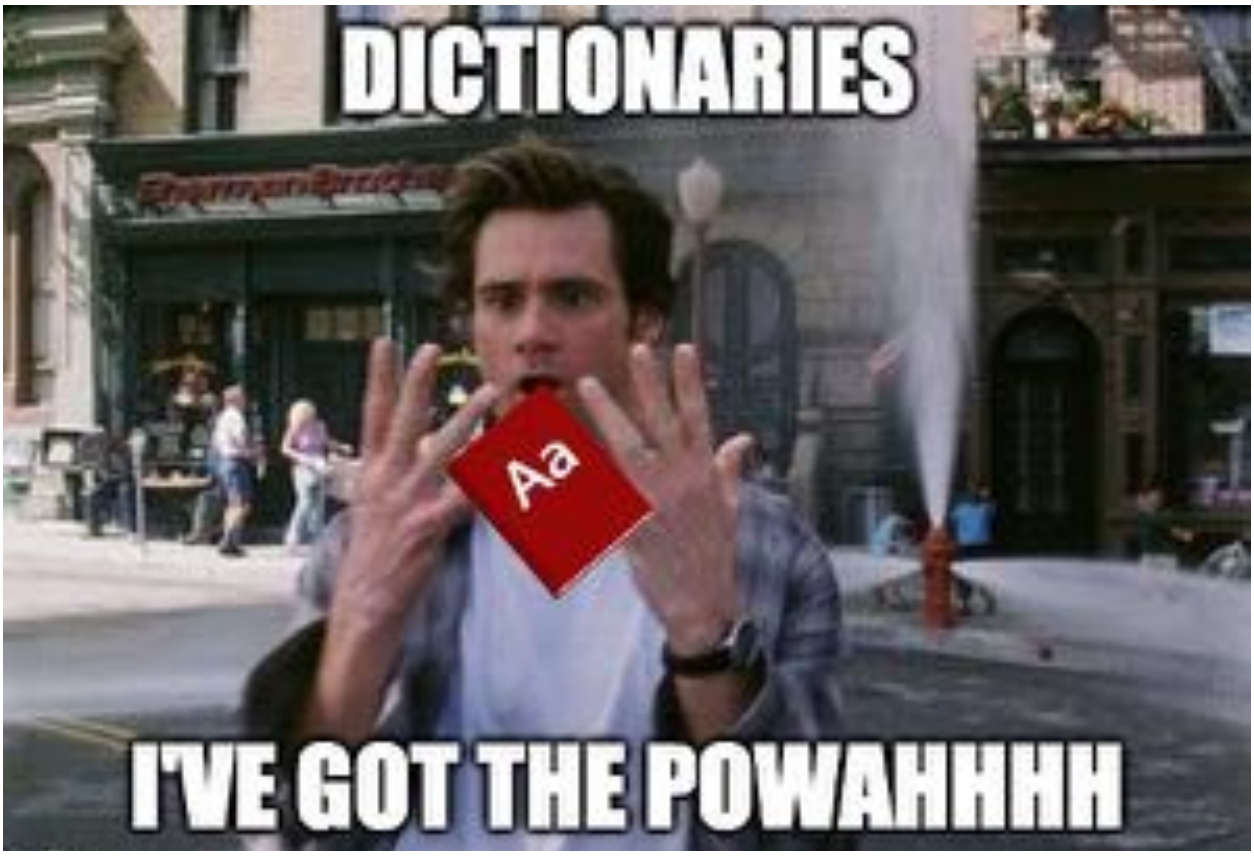
```
#0: Very
#1: few
#2: words
#3: here.
```

- In our last line of the program, we print a concatenation that says, "*testfile.txt* contains 4 words".

And if you're feeling like this…like…same.

Dictionaries

WHAT ARE DICTIONARIES?



- Dictionaries associate a **key** with a **value**. (Key-value pairs)
  - A **key** is a unique identifier (so no two keys are the same)
  - A **value** is something you *associate* with the key
- Some examples of dictionaries in real life:
  - A phonebook is a **dictionary**: the *name* is a key, and the *phone number* is the value.
  - A dictionary is also a dictionary: the *word* is a key, and the *definition* is the value.
  - The US government is a dictionary: the *Social Security number* is a key, and *information about employment* is the value.

TO CATCH A ~~PREDATOR~~ DICTIONARY: IDENTIFYING DICTIONARIES IN PYTHON

- In Python, this is the format of a Dictionary and its individual key-value pairs:

```
name_of_dictionary = {key: value, key: value, key: value }
```

- **Dictionaries** start and end with **curly braces** `{ }`
- **Key** and **Value** are separated by a **colon** `:`
- Each key-value **pair** is separated by a **comma** `,`

- Here's an example of a dictionary:

```
ages = { 'Chris': 32, 'Brahm': 23, 'Mehran': 50 }
```

- We have a dictionary called `ages` and three key-value pairs inside.
  - The first pair has a **key** called '`Chris`' and a **value** of `32`
  - Second pair has a **key** called '`Brahm`' and a **value** of `23`
  - Third pair has a **key** called '`Mehran`' and a **value** of `50`

- Key-value pairs don't always have to be string-int types; here's another example of a dictionary where both key and value are ints:

```
squares = { 2: 4, 3: 9, 4:16, 5:25 }
```

- Another example:

```
phone = { 'Pat': '555-1212', 'Jenny': '867-5309' }
```

  - Both the keys and values are strings here (we make the numbers also strings here because we want to keep the dashes in.)
- We can even have empty dictionaries:

```
empty_dict = { }
```

  - These come in handy when we want to *add* key-value pairs to a dictionary as we iterate through **`for`** loops and so on, similar to how we did this with empty lists.

ACCESSING ELEMENTS, SETTING VALUES, & UPDATING SHIT

- Recall our `ages` example from above:

```
ages = { 'Chris': 32, 'Brahm': 23, 'Mehran': 50 }
```

  - Think of the dictionary like a *set of variables* that are indexed by their *keys*.
  - In this case, the keys are '`Chris`', '`Brahm`', and '`Mehran`'
  - Use bracket notation `[ ]` and place the index (the ***key***) inside the brackets to access the ***value***.
  - For example, if we want to access the `'Chris': 32` key value pair, we would type:

```
>>> ages['Chris']
32
```

  - And the console would print the value associated with the key.
  - Because keys act like variables, w**e can also *set* values associated with the keys**:

```
>>> ages['Mehran']
50                  # prints current value
>>> ages['Mehran'] = 18 # set new value
>>> ages['Mehran']
18                  # prints updated value
```

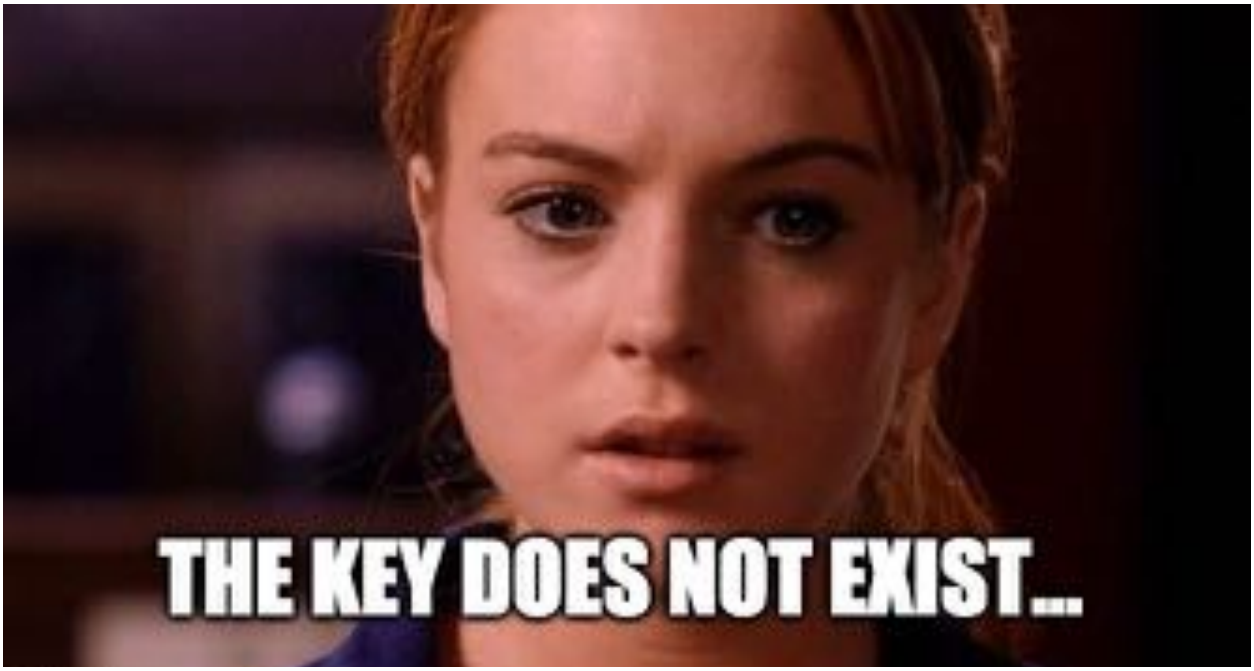
  - You can also ***update* values** using operands:

```
>>> ages['Mehran']
18                  # prints value
>>> ages['Mehran'] += 3 # update value!
>>> ages['Mehran']
21
```

- You can refer to the elements inside the dictionary by indexing using the key to get to or assign to or update the value associated with it.

CHECKING FOR KEYS

- If you try to access a key that doesn't exist, you will get a **KeyError**.



- To check if a key is in a dictionary, use the following notation:

```
>>> name_of_key in dictionary_name
```

- If you type that into your console, it will return a Boolean value. `True` if the key exists in the dictionary, and `False` if it does not.
- You can also do it the opposite way to see if a key *does not* exist in the dictionary:

```
>>> name_of_key not in dictionary_name
```

- These are useful inside `if` statements and `while` loops

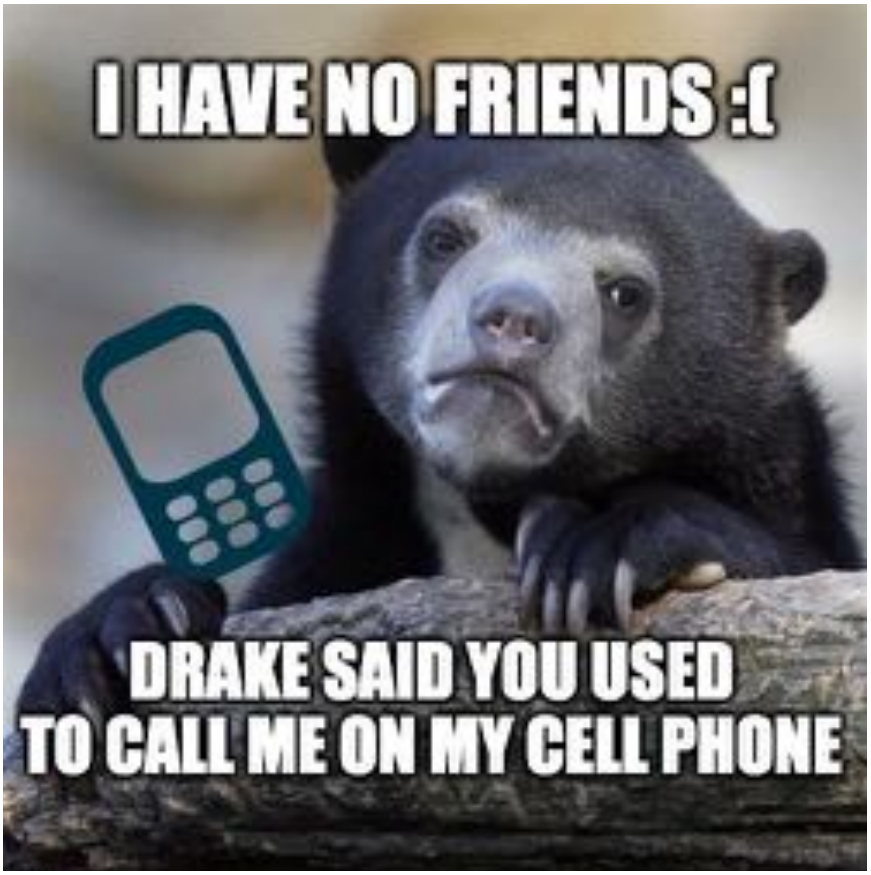
BUILD A DICTIONARY

- Remember how we said we could create an empty dictionary above? We can also add key-value pairs to them.

- Take this for example:

```
phone = { }
```

- This is an empty dictionary called phone. It doesn't have any contacts (or key-value pairs) stored in them. Phone is sad. :(



- Let's add some contacts to Sad Bear's phone! :) **To add key-value pairs to a dictionary, follow this format:**

```
name_of_dictionary[key] = value
```

- Cool, cool, now Sad Bear won't be sad anymore because he'll have so many friends to call!

```
phone['Pat'] = '555-1212'
phone['TP'] = '123-4567'
phone['Jenny'] = '675-4585
phone['Charmin'] = '305-2960'
```

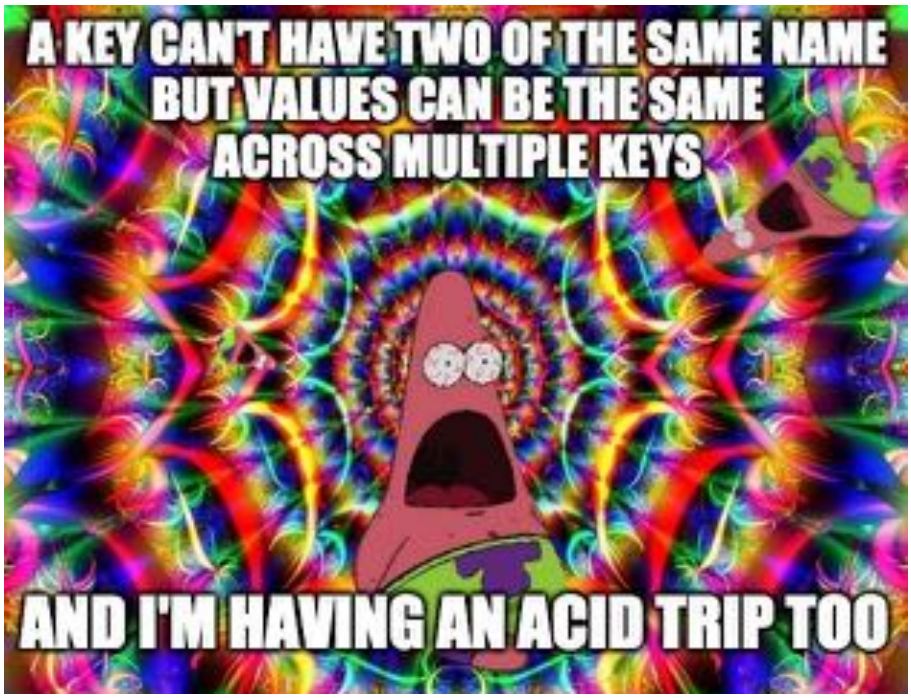
- Wao, so many friends! But what happens if we have another person named Pat?

```
phone['Pat'] = '000-0000'
```

- *OH NOEEEEEEES* we wrote over our other friend Pat's phone number! Remember that keys must be unique.
- Wow, Sad Bear's friends TP and Charmin got really friendly and moved in together. **We can update existing key-value pairs**. Charmin now has the same number as TP:

```
phone['Charmin'] = '123-4567'
```

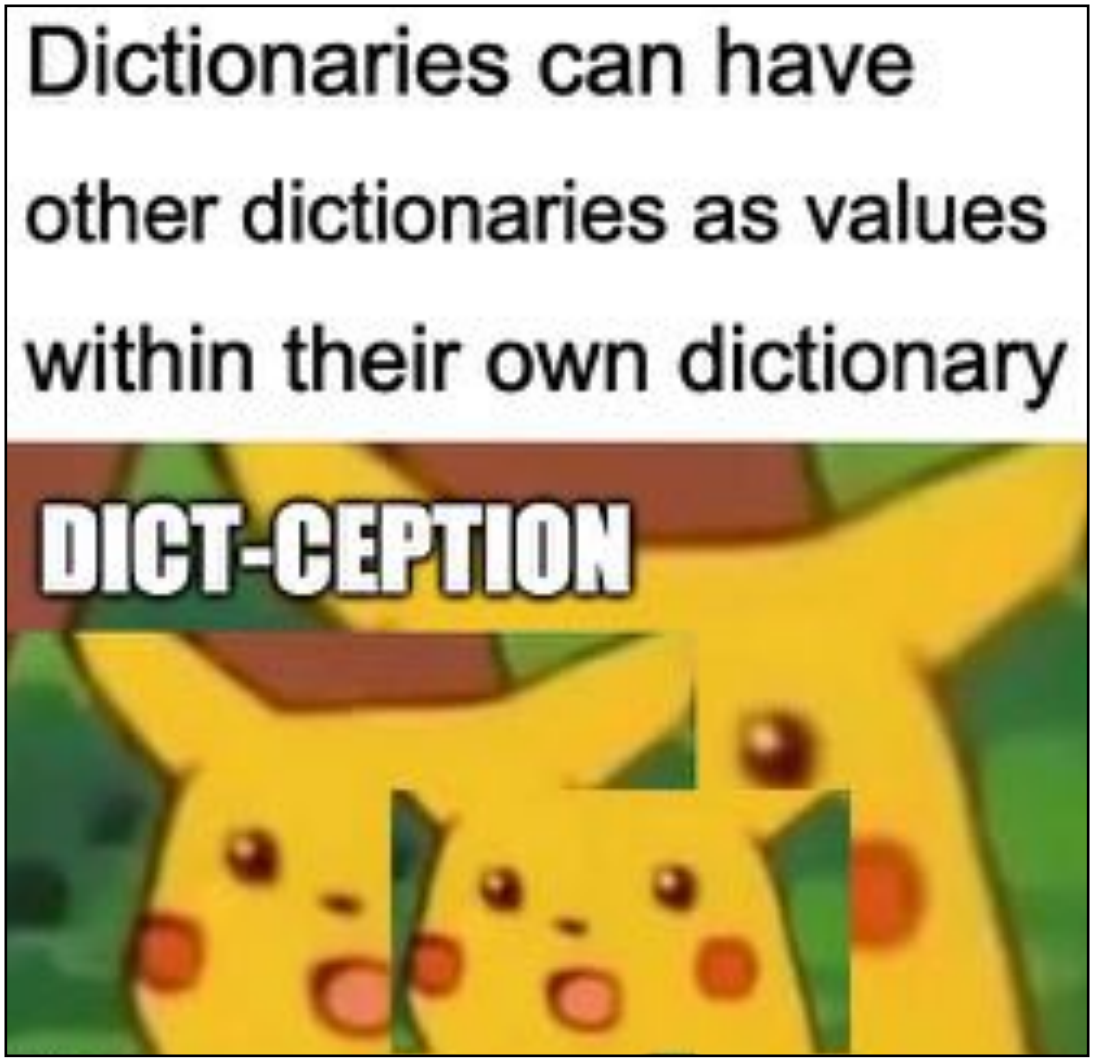
- Notice that unlike keys that have to be unique, we can have a key store the *same* value as another key.



Mutability and Dictionaries

KEY-VALUE AND DICTIONARY MUTABILITY

- **Keys** must be **immutable types** (primitives) that don't change internally. Examples include `int`, `float`, and `string`.
  - To change a key, you have to remove the key-value pair and add a new key-value pair with the new desired key.
- **Values** can be both immutable/mutable types. Examples include the primitives mentioned above, `lists`, and even `dictionaries`.
  - To change a value, you can reassigning a new value to the key.
- **Dictionaries** themselves are mutable. Therefore, changes made to a dictionary in a function persist after the function is done.



```
>>> def have_birthday(dict, name):
...  print("You're one year older, " + name + "!")
...  dict[name] += 1
...

>>> ages = {'Chris': 32, 'Brahm': 23, 'Mehran': 50}
>>> print(ages)
{'Chris': 32, 'Brahm': 23, 'Mehran': 50}

>>> have_birthday(ages, 'Chris')
"You're one year older, Chris!"

>>> print(ages)
{'Chris': 33, 'Brahm': 23, 'Mehran': 50}

>>> have_birthday(ages, 'Mehran')
"You're one year older, Mehran!"

>>> print(ages)
{'Chris': 33, 'Brahm': 23, 'Mehran': 51}
```

- `have_birthday()` will add a year to the age (stored in value) to the person whose birthday it is (stored in `name`).
  - `have_birthday()` has two parameters: `dict` (dictionary) & `name` (key).
  - When `have_birthday()` runs, it will `print`, "You're one year older, `name`!" (with the actual name concatenated in.)
  - Then, it will add 1 to the value associated with the `name`. It does this by accessing the *index* of `name` within the `dictionary` with this syntax: `dict[name]`
- Next, in our console: we define a `dictionary` and name it `ages`. We have three key-value pairs within `ages`: `'Chris': 32`, `'Brahm': 23`, and `'Mehran': 50`.
  - When we **`print`** `ages` to our console, notice it gives us the values exactly as they are. We haven't mutated them…yet.

- But next, we run `have_birthday()` and pass in `ages` as our `dict` and `'Chris'` as our `name`! Now, the console **`prints`**, "`You're one year older, Chris!`" and has increased his age (stored in the key's value) by 1.
- We see in the next line when we print `ages` again that it has now mutated `Chris's age`. He is now one year older!
- If we try this again on `Mehran`, we pass in `ages` and `'Mehran'` to `have_birthday()`.
- The console **`prints`**, "`You're one year older, Mehran!`" and has now increased his age by 1.
- Finally, when we **`print`** `ages` one more time, we notice that `Mehran's age` has been mutated and he is now one year older.

Notes by Regina Gerbeaux

Dictiona-Palooza! (Helpful Functions)

LET'S HAVE A DICTIONARY PARTY

```
ages = { 'Chris': 32, 'Brahm': 23,'Mehran': 50 }
```

Functionality	Format	Example	Output
Keys			

Returns value associated with key in dictionary. Returns "None" if key doesn't exist.	*dict*.**get(***key***)**	`print(ages.get('Chris'))`
		_
		`print(ages.get('Santa'))`
			`32`
			_
			`None`

Returns value associated with key in dictionary. Returns default if key doesn't exist.	*dict*.**get(***key*, *default_value***)**	`print(ages.get('Santa', 21))` ___ *sets default value to* `21` *for keys that don't exist*	`21`

Returns range of keys in dictionary	*dict*.**keys()**	`for key in ages.keys():`
		`  print(key)`
			`Chris`
			`Brahm`
			`Mehran`

Loop through dictionary using for-each loop. *We use for-each loops so often we don't have to append .keys() to our dict key.*	`for` **key** `in` *dict*:	`for key in ages:`
		`  print(key)`
		___
		notice how we don't have to append `.keys()` *to* `ages`*, unlike the loop above.*
			`Chris`
			`Brahm`
			`Mehran`

Returns list of keys in dictionary	**list(***dict*.**keys())**	`list(ages.keys())`	`['Chris', 'Brahm', 'Mehran']`

Functionality	Format	Example	Output
Values			

Returns range of values in dictionary	*dict*.**values()**	`for value in` **`ages.values():`**
		`  print(value)`
		———
		notice we have to append `.values()` *here, despite not needing to for* `keys` *above.*
			`32`
			`23`
			`50`

Returns list of values in dictionary	**list(***dict*.**values())**	`list(ages.`**`values()`**`)`	`[32, 23, 50]`
More Functions			
Remove key-value pair with the given key. **Returns** value from that key-value pair.	*dict*.**pop(***key***)**	`ages.`**`pop(`**`'Mehran'`**`)`** ——— *If we print* `ages` *again, it will print without Mehran's key-value pair.*	`50` ——— *returns value from the key you just popped off.*
Remove key-value pair from dictionary ***without*** returning anything	**del** *dict*[*key*]	**del** `ages['Mehran']` ——— *If we print* ages *again, it will print without Mehran's key-value pair.*	*(nothing returns or gets printed here)*
Removes *all* key-value pairs from a dictionary.	*dict*.**clear()**	`>>> ages.`**`clear()`** `>>> ages` ——— *leaves an empty dictionary!*	`{ }`
Get length of dictionary (number of key-value pairs)	**len(***dict***)**	**len**`(ages)` ——— *will check how many key-value pairs there are!*	`3`

Problem Solving: `count_each_word.py`

This program will counts the number of times a word appears in a text file. It uses a dictionary to store the results (where the key is a word, and the value is the number of times that word appears in the file.)

```
import sys

PUNCTUATION = '.!?,-:;'


def delete_punctuation(s):
     result = ''

     for char in s:
          if char not in PUNCTUATION:
          result += char
     return result


def get_counts_dict(filename):
    counts = {}

    with open(filename, 'r') as file:
        for line in file:
            words = delete_punctuation(line).split()
            for word in words:
                if word not in counts:
                    counts[word] = 1
                else:
                    counts[word] += 1
    return counts


def main():
    args = sys.argv[1:]

    if len(args) == 1:
        print(get_counts_dict(args[0]))
```

Step 1: `delete_punctuation( )`

```
import sys

PUNCTUATION = '.!?,-:;'


def delete_punctuation(s):
    result = ''

    for char in s:
        if char not in PUNCTUATION:
        result += char
    return result
```

- If you recall from the *beginning* of class at Strings, we already went through this whole piece of code together! It removes all punctuation marks from the text, and returns the punctuation-free version of the text. *For a more detailed explanation, please reference p.2 of notes.*

Step 2: `get_counts_dict( )`

```
def get_counts_dict(filename):
    counts = {}

    with open(filename, 'r') as file:
        for line in file:
            words = delete_punctuation(line).split( )
            for word in words:
                if word not in counts:
                    counts[word] = 1
                else:
                    counts[word] += 1
    return counts
```

- We give `get_counts_dict( )` one parameter: a `filename`
- Create an empty dictionary called `counts`, where the **word** is the **key** and the **number of times it appears** is the **value**.

- Use our `old` school method of opening `filename` into a Python-readable `file` using **`with`**.
  - Within `file`, loop through each **`line`** in the `file`, and:
    - Create List called `words` and set its value to `delete_punctuation()`, passing in the current `line` as the argument to remove all punctuation marks from the line.
    - Split off the carriage returns using `.split()`.
    - Then, loop through each `word` in our `words` List and see if it's already in our `counts` dictionary.
      - If it's not, add the `word` to the `counts` dictionary, and give it a value of `1`. If it is, increase the `word` counter by `1`.
- Once it has looped through the entire `file`, Python will close the `file` for us, and **`return`** the `counts` dictionary, where the key-value pairs are `word` instances and number of times they appear!

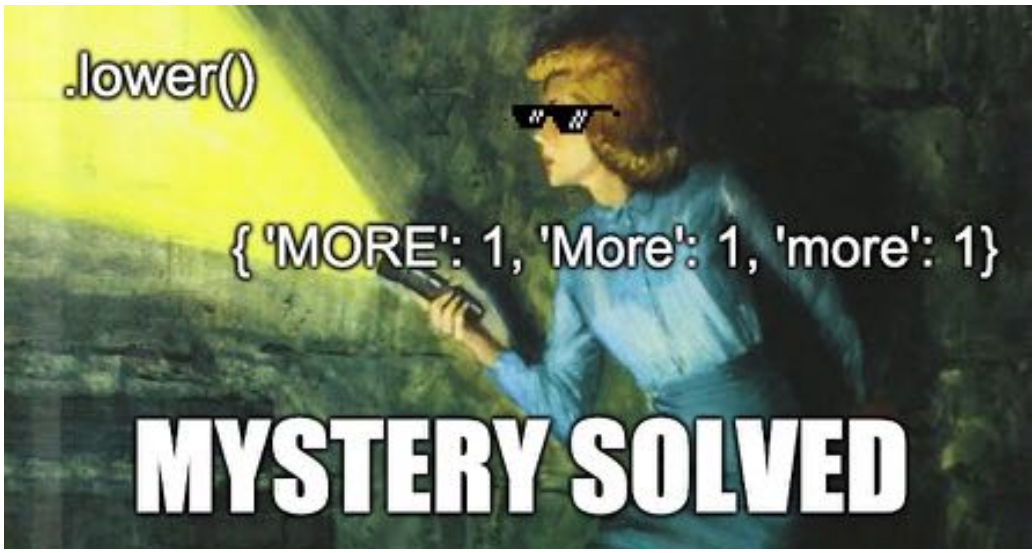
Step 3: `main()`

```
def main():
    args = sys.argv[1:]

    if len(args) == 1:
        print(get_counts_dict(args[0]))
```

- We have text that says `args = sys.argv[1:]`. This is because we have an **`import`** statement in the beginning that imports `sys`.
  - `sys` allows our program to access `sys.argv`, which accesses the arguments in the Command Line.
  - The name of the file we want to pass in should be the first and only command line argument; hence, we use the slice `[1:]`
  - Therefore, we include this line so that when we run our program, we can type the name of the program *followed by the name of the text file we want to pass in* and it can read the argument.

  - For example, if we go to the Terminal and run `count_each_word.py testfile.txt`, our Terminal would understand to run `count_each_word.py` and treat `testfile.txt` as an *argument*.

- **`Print`** the **`returned`** dictionary stored in `counts` using this code: `print(get_counts_dict(args[0]))`.
- The Terminal will print your full `counts` dictionary after that!
- **Note:** keys, when they are strings, are <u>case sensitive</u>. If the cases don't match, they are considered *different keys*.
  - Therefore, if you have a text file that has the word, "MORE" versus the word, "more", they will be counted as two different keys.
  - To bypass this, you could add an additional line of code that converts all text to lower case using the `.lower()` method.



Problem Solving: `phonebook.py`

This program lets us use a dictionary to maintain a phonebook.

```
def read_phone_numbers():
    phonebook = {}

    while True:
        name = input("Name: ")
        if name == "":
            break
        number = input("Number: ")
        phonebook[name] = number

    return phonebook


def print_phonebook(phonebook):
    for name in phonebook:
        print(name, "->", phonebook[name])


def lookup_numbers(phonebook):
    while True:
        name = input("Enter name to lookup: ")
        if name == "":
            break
        if name not in phonebook:
            print(name + " is not in the phonebook")
        else:
            print(phonebook[name])


def main():
    phonebook = read_phone_numbers()
    print_phonebook(phonebook)
    lookup_numbers(phonebook)
```

Step 1: `read_phone_numbers()`

```
def read_phone_numbers():
    phonebook = {}

    while True:
        name = input("Name: ")
        if name == "":
            break
        number = input("Number: ")
        phonebook[name] = number

    return phonebook
```

- `read_phone_numbers()` allows users to create entries for the `phonebook`.
  - It saves the `name` of the contact as the **key**, and the `number` of the contact as the **value**.
  - When user is done adding entries, it will return the `phonebook` dictionary to the caller.
- Create an empty dictionary called `phonebook`.
- Write a **`while`** loop that is always **`True`**. Within the loop:
  - Prompt user to input a `name`
  - Write an **`if`** statement that checks if user has entered a `name`.
    - If they don't, **`break`** the loop.
    - If they do enter a `name`, prompt user to input a `number`.
    - Set the entered `number` as the value to the key `name`.
  - Cycle through this loop until user does not enter a `name`.
- Once the loop is done running, return the `phonebook`, now filled with key-value pairs, to the function caller.

Step 2: `print_phonebook()`

```
def print_phonebook(phonebook):
    for name in phonebook:
        print(name, "->", phonebook[name])
```

- `print_phonebook()` prints all of the entries in a phonebook.
- `print_phonebook()` takes in the parameter `phonebook`.
- Write a **`for-each`** loop that loops through every `name` key in the `phonebook` dictionary.
  - Within the loop, the console will print each key-value pair using `print(name, "->", phonebook[name])` This results in the entries being printed in this format: `name -> number`
  - For example, if we have a key-value pair `{'Jenny': '675-4585'}` in our `phonebook`, it will print `'Jenny' -> '675-4585'` in the console.

Step 3: `lookup_numbers()`

```
def lookup_numbers(phonebook):
    while True:
        name = input("Enter name to lookup: ")
        if name == "":
            break
        if name not in phonebook:
            print(name + " is not in the phonebook")
        else:
            print(phonebook[name])
```

- `lookup_numbers()` allows users to look up a phone number in their phonebook by name.
- `lookup_numbers()` takes in the parameter `phonebook`.

- Write a **`while`** loop that is always **`True`**. Within the loop:
  - Prompt user to input a `name` to look up
  - Write an **`if`** statement that checks if user has entered a `name`.
    - If they don't enter a name, **`break`** the loop.
    - If they do enter a `name`, check if `name` is a key in `phonebook`.
      - If it isn't, **`print`**, "*`name`* `is not in the phonebook`."
      - If it is, **`print`** the **value** (`number`) associated with the `name`.

Step 4: `main()`

```
def main():
    phonebook = read_phone_numbers()
    print_phonebook(phonebook)
    lookup_numbers(phonebook)
```

- Let's put it all together!
- We create a variable called `phonebook` which will run `read_phone_numbers()` and prompt user to enter contacts. `phonebook`'s value will be the Dictionary returned from `read_phone_numbers()`.
- Next, we call `print_phonebook()` and pass in our now-populated dictionary `phonebook`. This function will print all of our entries.
- Finally, `main()` will execute `lookup_numbers()` and we pass in `phonebook`. This will prompt the user to look up the `number` of a contact in their `phonebook` by inputting the `name`.
