Unit - 3

Nested Functions

- Function defined inside another function.

> ℹ️ Whenever a function definition is seen by the interpreter, a *function entity* is created.

Scope and Lifetime of a variable

Scope

- of a variable indicates the accessibility of the variable in the program.
- Scope of a variable is confined to the level of indentations in which it was created and any level greater than itself. The variable is not accessible to code that has lesser indentation.

Lifetime

- of a variable indicated the time frame during which the variable and value is present in the memory.

Specifying input parameters

There are 5 different ways to specify parameters in a function:

1. Positional
2. Keyword
3. Default
4. Variable Length
5. Key:Value Pair

Positional

- Arguments passed in the same order they are defined in the function definition.

```python
def fun(a, b): #parameters
    print(a, b)

fun(10, 20) #arguments
# 10 → a
# 20 → b
```

Keyword

- Arguments are passed with the name of the parameters.

```python
def fun(a, b): #parameters
    print(a, b)

fun(b = 20, a = 10) # arguments
```

Default

- Parameter is defined to have a default value.
- This value is used if an argument for that parameter is not passed
- **Important** ℹ️ : A default argument should always be defined *after* all the positional arguments are defined.
- A default argument can be a variable too.

```python
def fun(a, b, c = 30):
    print(a, b, c)

fun(10, 20) # prints 10 20 30 | Default used.
fun(10, 20, 40) # prints 10 20 40
```

Screwed up program:

```python
def fun(a = [], b = {}):
    print(a)
    print(b)
    a.append(len(a))
    d[len(a)] = len(a)

fun()
fun()
fun()
fun([1,2,3], {4 : "four"})
fun()

'''
output:
[]
{}
[0]
{1: 1}
[0, 1]
{1: 1, 2: 2}
[1, 2, 3]
{4: 'four'}
[0, 1, 2]
{1: 1, 2: 2, 3: 3}
'''
# The previous values are used for every function call!
# This happens for only mutable datatypes, like list, dictionary etc.
# BUTTTTT, If we do not use the default parameters, then previous values are not used. The previous values from the previous DEFAULT FUNCTION Call is used. Tell me that's not messed up
```

Variable Length

- Allows to use variable length arguments
- Uses a tuple for the arguments

```python
def fun(a, *btuple):
    print(a)
    print(btuple)

fun(10)
# output:
# 10
# ()

fun(10,20,30,40,50)
# output:
# 10
# (20, 30, 40, 50)
```

Q1. Write a python program to using functions which takes arbitrary number of strings (variable number of strings) and returns the length of the longest string.

Key-Value Pair

- To pass a variable-length key-value pair as an argument to a function, Python provides a feature called `**kwargs`
- *kwargs* stands for Keyword arguments. It's useful when you want to deal with variable length named arguments in a function.
- syntax:

```python
def functionName(**anything):
    statement(s)
```

- Example:

```python
def printKwargs(**kwargs): #the parameter name could be anything, not necessarily 'kwargs'
    print(kwargs)


printKwargs(Argument_1='arg1', Argument_2='arg2')

# output:
# {'Argument_1': 'arg1', 'Argument_2': 'arg2'}
```

Q2. Write a python program using a user defined function to sort the given numbers in ascending order.

Recursion

- is when a function calls itself.
- You need to think of

1. Terminating Condition
2. Each recursive call should be towards the terminating condition. Else, your function would keep calling itself infinitely

Advantages

- Makes your code look clean and small - Although less readable and understandable.
- Sequence generation is easier.
- A complex task can be broken down

> 💡 Writer's note: Recursive functions are really difficult to understand. Your code must be ***readable!***. It's very important while you are working on a project. Recursion also uses more memory. Please use an iterative approach instead of a recursive approach whenever possible, please. Thank you. Bye

Disadvantages

- Code readability is *royally screwed*. Even the most simple recursive functions are hard to follow and understand.
- They take up a lot of memory and time. They are computationally more expensive. You need your program to be quick and efficient!
- They are also hard to debug. If you don't know what's going on, how on earth will you *debug*

Q3. Write a python program to find GCD of 2 numbers using recursion.
Q4. Write a python program to print the fibonacci sequence. Take input once as the *number of terms* and once as the *max term*

> ℹ️ The `nonlocal` keyword is similar to `global` but it doesn't search for the variable in global scope, instead it checks in the environment directly just outside the function. This is especially useful in nested functions.

Pass by value and Pass by reference

- You might want to punch something after reading ahead, so brace yourself. Python's argument passing model is **neither "Pass by Value" nor "Pass by Reference" but it is *"Pass by Object Reference"*.**
- Look at the below program

```python
#Acts like Pass By Value
def set_list(my_list):
    my_list = ["A", "B", "C"]
    print(id(my_list)) # won't be same as that outside


#Acts like Pass By Reference
def add(my_list):
    my_list.append("D") # will be same as that outside


my_list = ["E"]
print(id(my_list)) # outside memory ID

set_list(my_list)
print(my_list) # ['E']
add(my_list)
print(my_list) # ['E', 'D']
```

- In the `set_list` function, we are redefining the list and hence it acts like ***Pass By Value***. What this means is that only the value is passed and it is completely disconnected to the original list outside the function
- In the `add` function, we are just appending to the same list and so python treats it as ***Pass By Reference***. This means that the value **IS** connected to the list outside and whatever modifications we make here will be reflected outside too.
- Yeah that's a pain in the 🍑
- Python is weird 🫠 Use C.
- **ALSOO**, This appending thing having the same memory address as outside happens only for *mutable* datatypes. *immutable* datatypes always behave as **Pass By Value** only : ie, only value is passed. Outside value is not modified.
- Above point is demonstrated here:

```python
def func(a):
    a += ' anurag'
    print(a) # hello anurag

#using global
def globalAdd():
    global a
    a += ' anurag'
    print(a) # hello anurag


a = "hello"
print(a) # hello
func(a)
print(a) # hello
globalAdd()
print(a) # hello anurag
```

- Here, in the `func` function, we are trying change a string, which is *immutable*, so even if we are just modifying, it still treats it as a separate variable inside and does not touch the variable outside.
- However, this behaviour can be modified by using the `global` keyword. This makes it use the variable `a` from the **global scope**, ie, outside the function. So whatever we do to `a` here will be reflected outside.
- Using `global` seems like the way to go, it avoids so much confusion, right? **wrong!** When you have multiple functions in your program, it becomes super difficult to track what variable is changed where. Now imagine multiple people working on multiple functions. This makes it difficult to *debug*. In normal people words - it makes it difficult to find where shit went down when your program isn't working. It defeats the entire purpose of using a function in the first place.

Function Callback

- Function calls another function
- *Passing the function itself as an argument*
- **Different from recursion**
- More like an API

Advantages:

- Calling function can call the callback function as many times as it needed (doesn't really make sense but was given in the slides)

- Calling function can pass appropriate parameters according to the task to the called functions - This allows information hiding.

- The calling function acts as an interface through which other functions can be called.

- A good example of callback is the *map function*. The map function maps (goes through) every element in the input list and *passes it through the int function.* The output produced should be a list with all the elements as an int

```python
l = list(map(int, ['1', '2', '3']))
print(l)

# output:
# [1, 2, 3]
```

- More understandable example:

```python
def getSquare(x):
    return x**2
def getRoot(x):
    return x**0.5

def masterFunction(x, givenFunction):
    x += 2
    x = givenFunction(x)
    return x

print(function(2, getSquare)) # 16
print(function(2, getRoot)) # 2.0
```

Main Advantage

is that you can use one function as a master function, and make the function behave differently depending on what function we pass to it as a parameter.

- Here, the function called `masterFunction` either gets a square or gets a root depending on what function we pass as an argument to it.

Q5. Write a python program using callback function to define 4 functions - triangle, rectangle, square, draw.

Closures

- A closure is *a function object* that *remembers values* in enclosing scopes even if they are not present in memory.
- We'll take an example,

```python
def transmit_to_space(message):
  # This is the enclosing function
  def data_transmitter():
      # The nested function
      print(message)
  return data_transmitter

fun2 = transmit_to_space("Burn the Sun!")
fun2()

fun3 = transmit_to_space("Hello Aliens!")
fun3()

# output:
# Burn the Sun!
# Hello Aliens!
```

- Now here, we are returning the `data_transmitter()` function. Notice that this return statement belongs to the `transmit_to_space()` function.
- We call this function twice, with 2 different `message`s.
- These messages are printed using the `data_transmitter()` function, BUT, the `data_transmitter()` function is *not* called inside the `transmit_to_space()` function.
- Now that we are returning the function, we can store this inside an *object* or a *variable*. We are calling this function twice and storing it in the `fun2()` and `fun3()` objects.

> ℹ️ Everything in python is an object. The terms `object` and `variable` is used interchangeably here.

- Now that we have these objects, we can call them.
- Both these objects remember the `message` that was function while calling the `transmit_to_space()` function.
- This whole confusing shenanigan is called a ***Closure***

Where do we use this then?

Closures can avoid use of global variables and provides some form of data hiding. (Eg. When there are few methods in a class, use closures instead).

> Remember when I said using global variables causes a mess, yes, this is what I was talking about. **AVOID GLOBAL VARIABLES AT ALL COSTS**. I have trauma with it 😢

- One thing to keep in mind is that you can mix variables up, both local and the ones called outside with this Closure thing. I bet profs teach this topic to just make confusing ISA questions and make us loose marks :\
- Code readability is everything in a software organisation. While this is better than using global variables, it's still not a 100% simple to read and understand.

Decorators

- It allows you to modify the behaviour of a function or class.
- It allows you to wrap another function in order to *extend* the behaviour of a wrapped function, without permanently modifying it.
- Example:

```python
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def ordinary():
    print("I am ordinary")
```

- Now if you run the following in a shell:

```python
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

- The function `ordinary()` got decorated and the returned function was given the name `pretty`.
- We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).
- Python has a syntax to simplify this. We can use the `@` symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```python
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to

```python
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

What if your function had parameters?

- Let's have another example,

```python
def divide(a, b):
    return a/b
```

- If we pass `b = 0` to the above program, it will return a `divide by zero` error.

```python
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

- Now we'll make a decorator to check for this condition:

```python
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner


@smart_divide #← Notice the @ syntax
```

```python
def divide(a, b):
    print(a/b)
```

- This new implementation will return `None` if the error condition arises:

PYTHON
```python
>>> divide(2,5)
I am going to divide 2 and 5
0.4

>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

- You might have noticed that parameters of the nested `inner()` function inside the decorator is the same as the parameters of functions it decorates. Taking this into account, now we can make general decorators that work with any number of parameters
- In Python, this magic is done as `function(*args, **kwargs)`. In this way, `args` will be the tuple of positional arguments and `kwargs` will be the dictionary of keyword arguments.

> ℹ️ Scroll Up and visit the variable length arguments to a function section if you aren't sure how `*args` and `**kwargs` work.

It would work something like this:

PYTHON
```python
def works_for_all(func):
    def inner(*args, **kwargs):
        print("I can decorate any function")
        return func(*args, **kwargs)
    return inner
```

Chaining Decorators

- Multiple decorators can be chained in Python.
- This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

PYTHON
```python
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner



def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner


@star
@percent
def printer(msg):
    print(msg)


printer("Hello")
```

- Output:

```shell
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

- The above syntax of

```python
@star
@percent
def printer(msg):
    print(msg)
```

is equivalent to

```python
def printer(msg):
    print(msg)
printer = star(percent(printer))
```

- This decorator concept is a little tricky to wrap your head around. You can refer to these articles to understand them: this and this

---

Q6. Write a python program using decorated functions which extends the features to calculate average marks of 3 people.

Important Programs

Q1. Write a program using functions that accepts a string and calculates the number of uppercase and lowercase letters.
Q2. Write a python program using functions that accepts 2 strings as input and prints/or returns the string with maximum length as the output. If same length => return or print both the strings.
Q3. Write a python program using functions that takes a list as an argument and returns a new list

with the duplicate values removed.
Q4. Write a python program using functions to achieve the following:

```python
a = ['karnataka', 'tamilnad', 'karnataka', 'karnataka', 'karnataka', 'tamilnad',
'kerala']
b = ['mysore', 'chennai', 'hassan', 'shimoga', 'madurai']
output: {dictionary of 'a' mapped to 'b', no particular order}
```

Q5. Write a python program to reverse a string using recursion

---

GUI | Graphical User Interface

- Every application can not be run on a terminal. We need a graphical user interface.
- Python provides a package called `tkinter`(pronounced Tea-Kay-inter)
- `tkinter` is used to make a GUI.
- Tkinter actually comes along when we install Python. While installing Python, we need to check the td/tk and IDLE checkbox. This will install the tkinter and we need not install it separately. However, this is only on windows - On Linux and macOS, we need to `pip install tkinter`

How to use `tkinter?`

- **Create a window**

```python
import tkinter as tk
root = tk.Tk() # creates window
root.mainloop()
```

- `window.mainloop()` tells Python to run the Tkinter **event loop**. This method listens for events, such as button clicks or keypresses, and blocks any code that comes after it from running until you close the window where you called the method.
- If you do not include the `.mainloop()` at the end of your program, the tkinter application will **not** be displayed.
- However, this is not the case if you are programming on the python shell (interactive mode). The window is displayed as soon as you enter `root = tk.Tk()`

> ℹ️ If you import tkinter as `from tkinter import *`, it loads all functions of tkinter into the global scope. This means that you can directly do `root = Tk()` and it will work. There is no need of typing `tk.<widget>`. This is not recommended though. **Using `import *` in python programs is considered a bad habit** because this way you are polluting your namespace, the import * statement imports all the functions and classes into your own namespace, which may clash with the names of the functions you define or the names of the functions of other libraries that you import.

- Every element in tkinter is a *widget*

- To make an element, we
  1. Create a widget.

2. Add it to the window.

How To Make A Label?

PYTHON
```python
# syntax:
# object = tk.Label(<window object>, <text>)
label1 = tk.Label(root, text = "Hello")
```

How To Make A Button?

PYTHON
```python
# syntax:
# object = tk.Button(<window object>, <text>, <command (function) to be executed
on click>)

def onClickFunction():
    print("Clicked")

button1 = tk.Button(root, text = "click me", command = onClickFunction)
```

Taking input?

PYTHON
```python
# syntax:
# entry1 = tk.Entry(<window object>)

entry1 = tk.Entry(root)
string1 = entry1.get() # gets whatever you've entered
print(string1)
```

- Well so far we've only figured out how to put widgets on our screen. What about if we want to modify *where* in the window we want to place these widgets?

Frames

- Frames work as a container for widgets you want to place inside them

PYTHON
```python
import tkinter as rk
root = tk.Tk()
screenLayout = tk.Frame()
screenLayout.pack()
root.mainloop()
```

> ℹ️ Another way of putting widgets on the screen is `widget.pack()`. It's a little bit different than just giving the `root window` widget at the time of creation though (something like `button1 = tk.Button(root, ...)`). This makes the window shrink to `pack` only the widgets inside it.

- Now that we have created our frame, we can pack widgets inside it.

```python
button1 = tk.Button(master = "screenLayout", text = "hello", command = onClick)
screenLayout.pack()
```

- This way you can make multiple frames on the screen to have multiple *localised* grids. This allows you to place your widgets where you want.
- I won't go into much detail into how because this was never taught in class, although it is there on the slides :\ When I asked the prof, he said 'it's very simple, just go through them and ask me if you get any doubts. I'm always there, just text me' 🥲
- Anyways, let me explain the `.pack()` geometry manager a little more

The `.pack()` geometry manager

- `.pack()` places your widgets inside the parent *frame* or *window* in a packed way. It basically shrinks the parent element(frame or window) to the elements inside it.

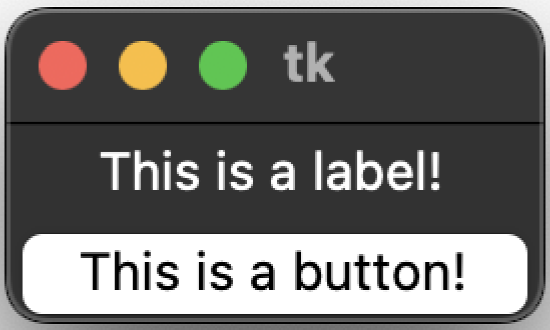
```python
import tkinter as tk

root = tk.Tk()
def onClick():
    pass
label1 = tk.Label(root, text = "This is a label!")
button1 = tk.Button(root, text = "This is a button!", command = onClick)

label1.pack()
button1.pack()

root.mainloop()
```

creates a little window like this:



- Notice how everything is packed and the window is *shrunk* to only fit the widgets inside it
- Well this isn't very convenient, so there are other geometry managers too

The `.place()` geometry manager

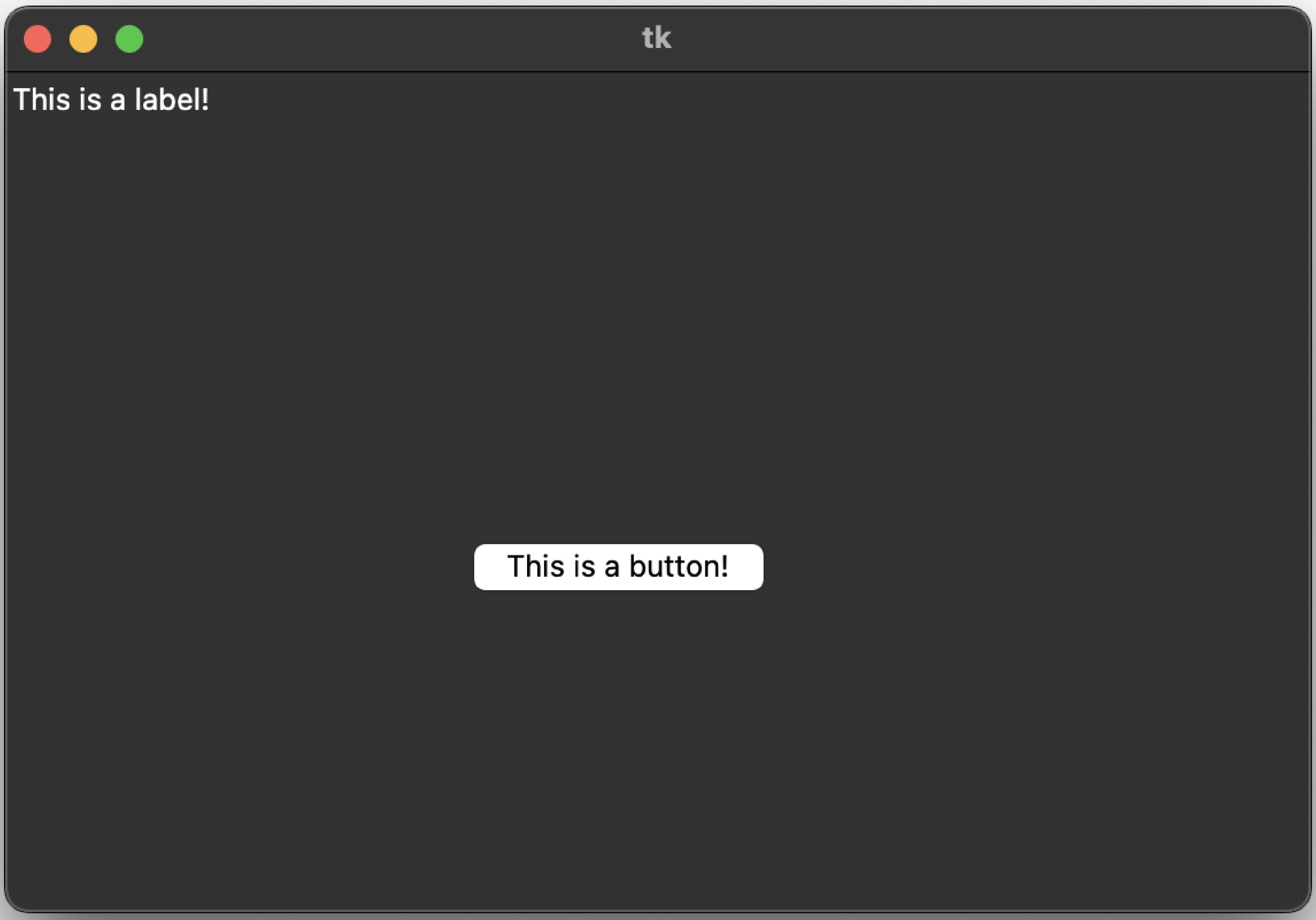
- The `.place()` takes in two keyword arguments : x and y
- These would be the x and y positions of the widgets *in pixels*

```python
import tkinter as tk

root = tk.Tk()
def onClick():
    pass
label1 = tk.Label(root, text = "This is a label!")
button1 = tk.Button(root, text = "This is a button!", command = onClick)

label1.place(x = 0, y = 0)
button1.place(x = 200, y = 200)

root.mainloop()
```



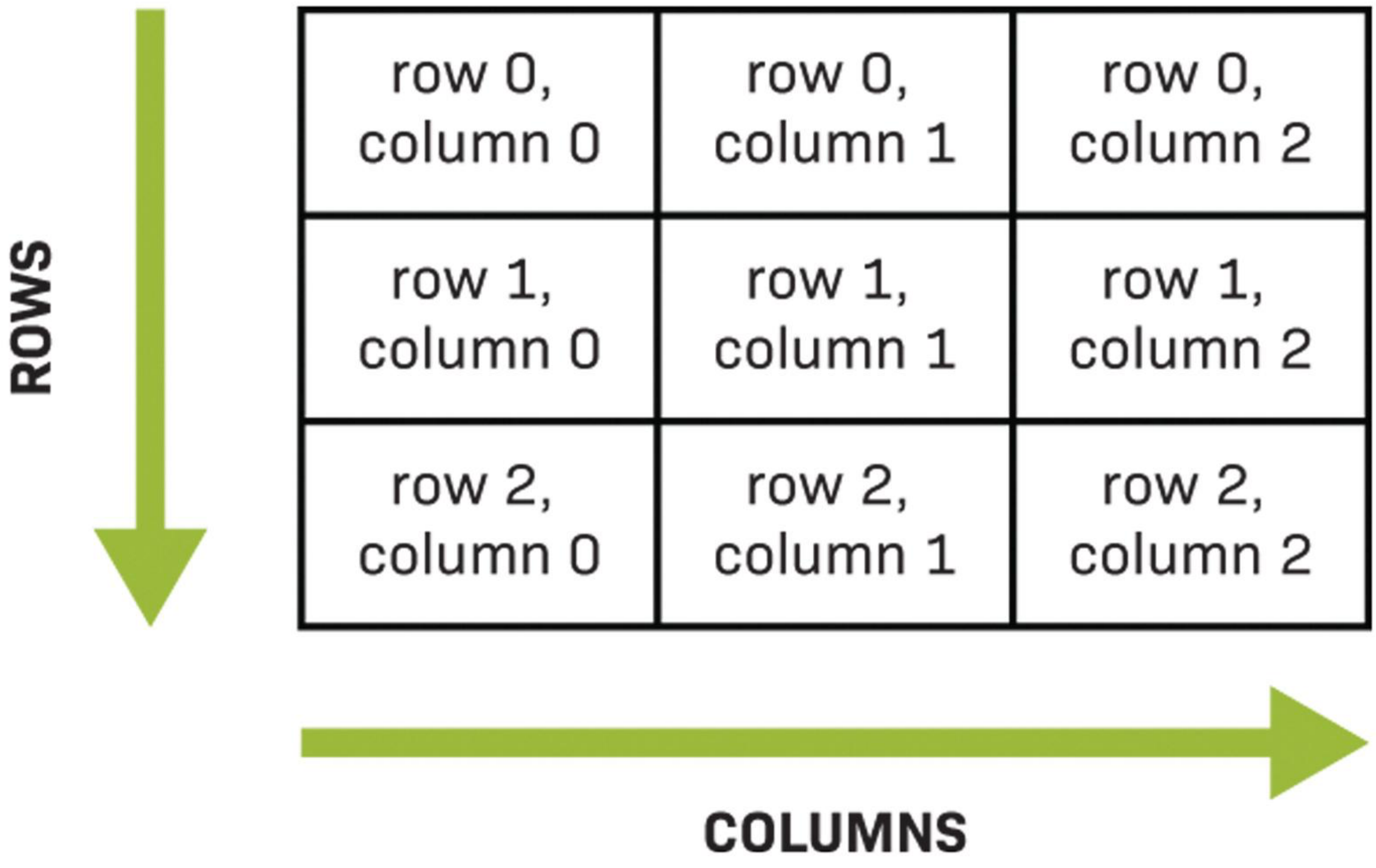
- Still not that convenient, who on earth would sit and count pixels?

The `.grid()` geometry manager

- This is what most programmers use. It provides the power of `.pack()` in a format that is much easier to understand and maintain.

- Rows and Columns basically.

- The syntax works like this: `element.grid(<row>, <column>)`



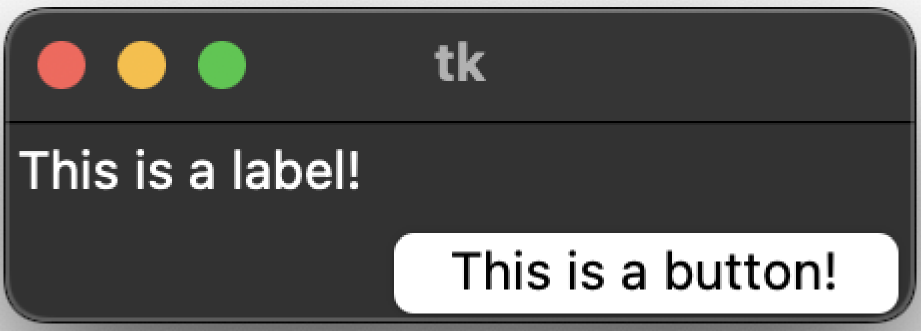
- Let's take our usual example:

```python
import tkinter as tk

root = tk.Tk()
def onClick():
    pass
label1 = tk.Label(root, text = "This is a label!")
button1 = tk.Button(root, text = "This is a button!", command = onClick)

label1.grid(row = 1, column = 1)
button1.grid(row = 5, column = 10)

root.mainloop()
```



- Notice how this is a combination of `.pack()` and the grid properties. It also makes the window fit the elements, but also gives you rows and columns to work with.

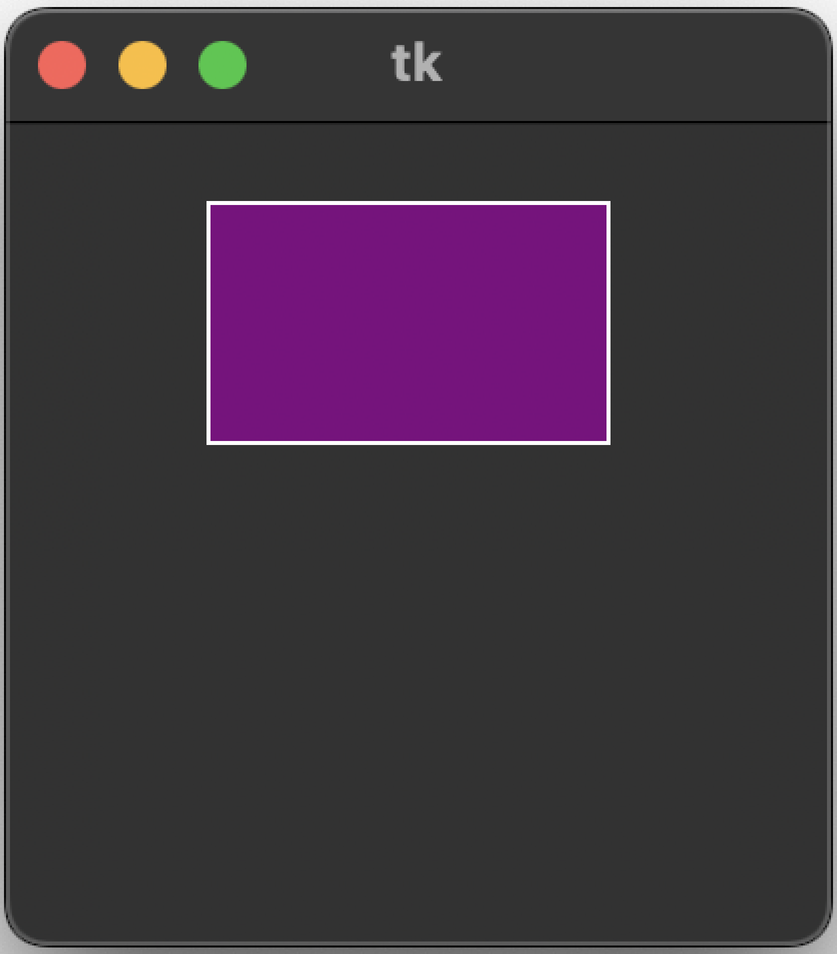
How do you draw something though?

- This is a question nobody asked but it's there in the syllabus
- You can use this thing called a `canvas`
- Works like this:

```python
import tkinter as tk

root = tk.Tk()
canvas1 = tk.Canvas(root, width = 200, height = 200)
canvas1.pack()
canvas1.create_rectangle(50, 20, 150, 80, fill = "purple") # the numbers 50,
20... are the coordinates of the 4 corners of the rectangle, in pixels.


root.mainloop()
```



> 💡 Writer's note: tkinter is kinda ugly. Let's face it. It isn't super easy either. I'd suggest anybody serious about GUI programming to use the `Qt` (pronounced 'cute') framework instead of the `tk` framework. `Qt` framework, as the name suggests, is cute. It was named that way [intentionally](). It has a *drag and drop* style editor to create and place your widgets AND you can define `css` styles for every widget that you create. It's MUCH easier to work with and looks way better! Visit the [Docs]()

Made with ♡ by Anurag Rao
📞 9663006833 for suggestions and corrections