String Indexing, Slicing, and Looping

15-110 – Wednesday 09/17

Quizlet

Pencils in the front!

Learning Goals

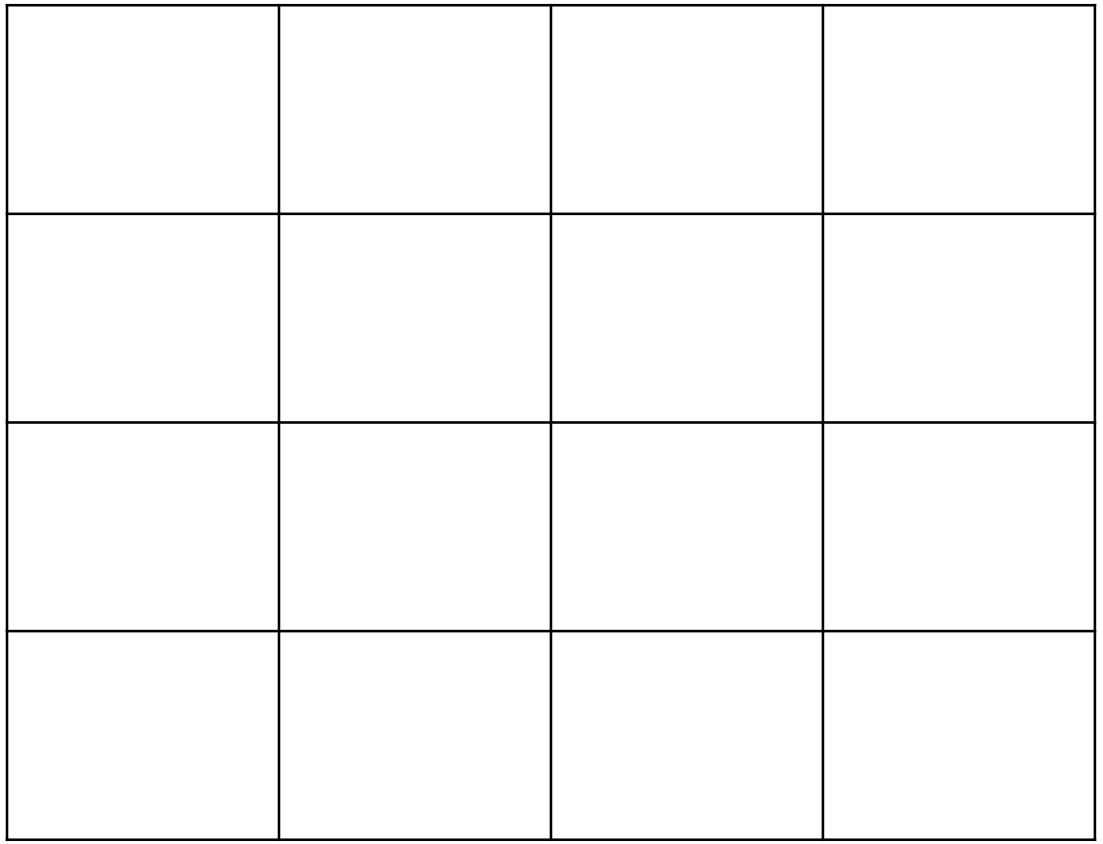
- **Index** and **slice** into strings to break them up into parts
- Use for loops to loop over strings by **index**

Revisiting Drawing Grids

We can use nested `for` loops to **draw a grid**!

We'll use nested `for` loops (along with math and logic) to determine where to draw each square with `create_rectangle`.

A grid with `gridSize`=4, has 4 columns and 4 rows of equal sized squares.



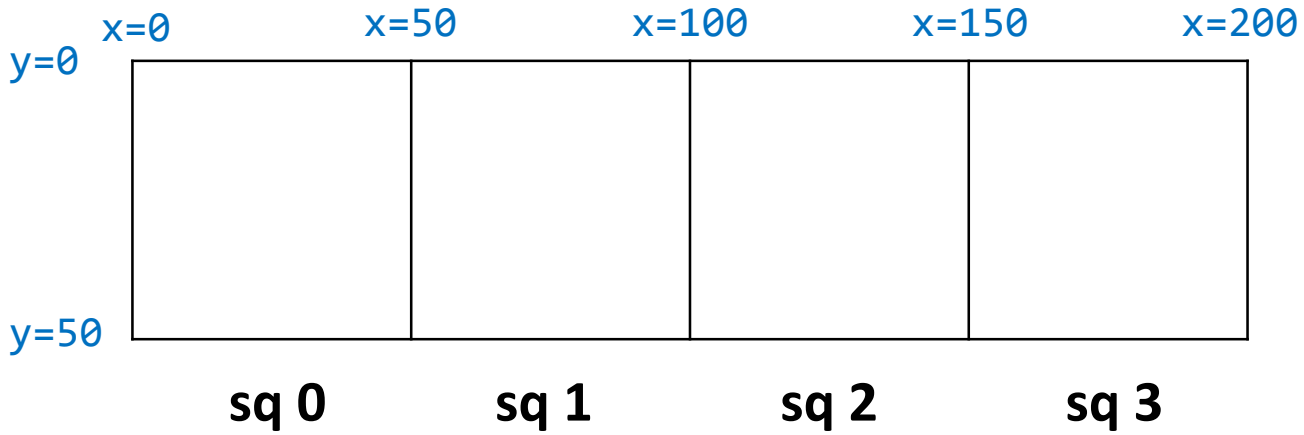
To draw a grid, we need to **figure out the pattern** to draw 50x50 squares.

What is the **loop control variable** and what is the start, stop, and step?

For each rectangle, what are the **topX and topY coordinates** based on the control variable?

For each rectangle, what are the **bottomX and bottomY coordinates** based on the control variable?

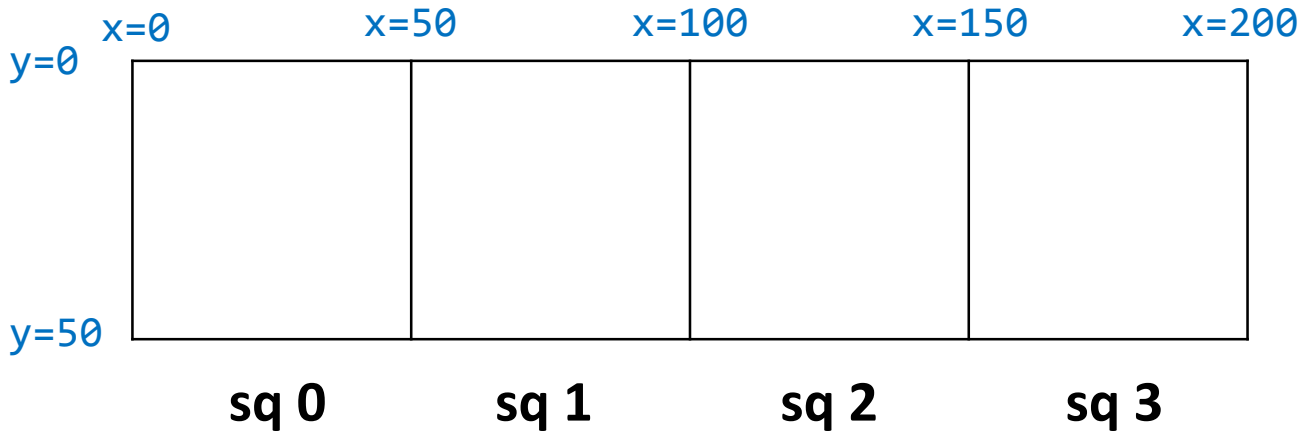
Desired outcome:



To draw a grid, **we loop over the columns**, the left side is the square number * the square size.

```
def drawGrid(canvas, gridSize):
  for col in range(gridSize):
      topX = col * 50
      bottomX = topX + 50
      topY = 0
      bottomY = 50
      canvas.create_rectangle(topX,
                              topY,
                              bottomX,
                              bottomY)
```

Desired outcome:



We can draw each row by putting the **logic for drawing a row inside an outer loop**.

The outer loop represents a cell's row, while the inner loop represents a cell's column.

Calculate the top of each cell based on the value's row, using the same logic that found the column coordinates.

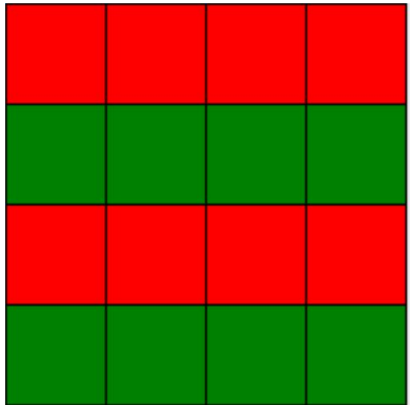
```
def drawGrid(canvas, gridSize):
    for row in range(gridSize):
        for col in range(gridSize):
            topX = col * 50
            bottomX = topX + 50
            topY = row * 50
            bottomY = topY + 50
            canvas.create_rectangle(topX,
                                    topY,
                                    bottomX,
                                    bottomY)
```

We can **add stripes with conditionals**.

We can make the grid more exciting by adding colors to the cells, to draw stripes.

Stripes alternate by row. Check whether the row is odd or even using the mod operator.

```
if row % 2 == 0:
    color = "red"
else:
    color = "green"
canvas.create_rectangle(topX,
                        topY,
                        bottomX,
                        bottomY)
```



String Indexing and Slicing

What we already know: Text values are called **strings**.

Text is recognized by Python as a string by putting it into either **single quotes**: `'Hello'` or **double quotes**: `"Hello"`

Strings can be **concatenated** using addition operator `+`:

```
>> "Hello" + "World"
HelloWorld
```

Strings can be **repeated** using multiplication operator `*`:

```
>> "Hello" * 3
HelloHelloHello
```

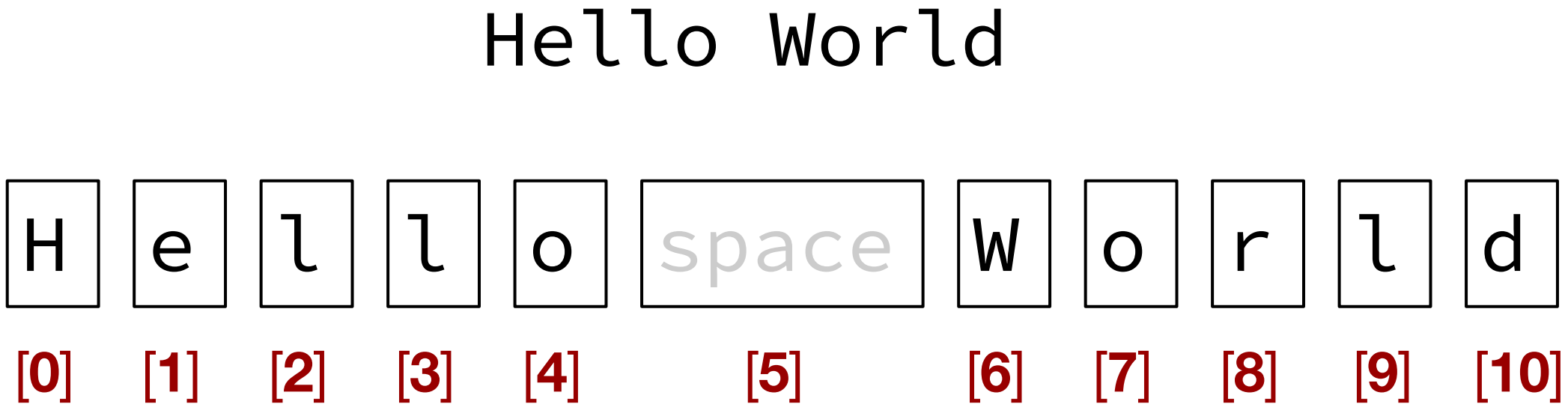
We represent text as **individual characters**.

Break text into characters:

```
Hello World
```

We can manipulate text data in many ways by doing operations with these individual characters!

Each character in a string is stored at a specific location **(index)**.



Indices **start at 0**!

Indexing is when we use **square brackets** to get a character at a specific location.

```
s = “KIMCHEE”
c = s[2] #M
```

We can get the number of characters in a string with the built-in function `len(s)`.

```
print(len(s)) #7
```

There are some **common string indexes**.

How do we get the first character in a string?

```
s[0]
```

How do we get the last character in a string?

```
s[len(s) - 1]
```

What happens if we try an index outside of the string?

```
s[len(s)] # runtime error
```

Activity: What is the index?

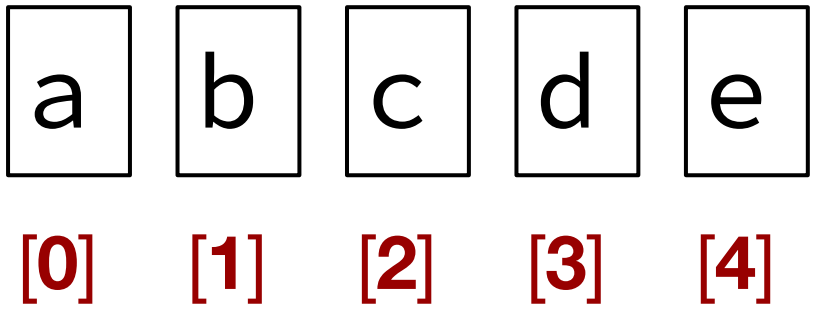
Given the string `"abc123"`, what is the index of...

`"a"`?

`"c"`?

`"3"`?

We can get a subset of the characters of a string using **slicing**.



Slices are similar to ranges (stop is not inclusive, start and step are optional) but the syntax is **inside square brackets and separated by colons** **[**<*start*>**:**<*stop*>**:**<*step*>**]**

```
s = "abcde"
print(s[2:len(s):1])    # prints "cde"
print(s[0:len(s)-1:1]) # prints "abcd"
print(s[0:len(s):2])    # prints "ace"
```

Slices have 3 parts, **ALL are optional** with default values.

With **no colons, we are indexing** and only getting one character

`s[<index>]`

To specify a **start we need 1 colon**, default value is 0

`s[<start>:<stop>]`

To specify a **step we need 2 colons**, default value is 1

`s[<start>:<stop>:<step>]`

When slicing using colons, **stop is also optional (not inclusive)** and has a default value of `len(s)`.

Slices have 3 parts, **ALL are optional** with default values.

When we want to use a default value for a part, **we leave it blank in the slice**.

`s[::]` and `s[:]` are both the string itself, unchanged

`s[1:]` is the string without the first character (start is `1`)

`s[:len(s)-1]` is the string without the last character (end is `len(s)-1`)

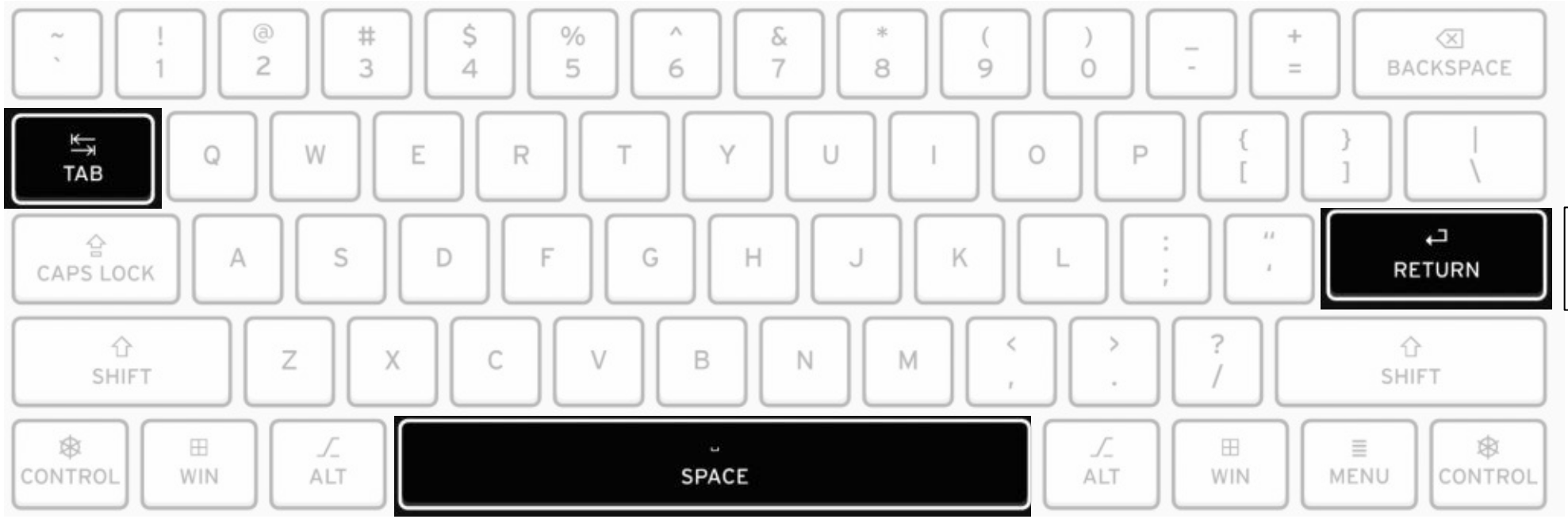
`s[::3]` is every third character of the string (step is `3`)

Activity: Find the Slice

Given the string `"abcdefghij"`, what slice would we need to get the string `"cfi"`?

More String Things

Whitespace is represented in different ways in Python depending on the character.



The \ character is a special character that indicates an **escape sequence**. It is modified by the letter that follows it.

These two symbols are treated as a single character by the interpreter.

```
"ABC\nDEF" # '\n' = newline, or pressing enter/return
"ABC\tDEF" # '\t' = tab
```

Haiku by Carol A. Coiffait

You can use **triple quotes** to make multi-line strings.

```
s = """This Autumn midnight
Orion's at my window
shouting for his dog."""
```

is equivalent to:

```
"This Autumn midnight\nOrion's at my window\nshouting for his dog."
```

We **check if a character or substring is within a string** using the `in` (and the opposite `not in`) operator.

```
"e" in "Hello" # True
"W" in "CRAZY" # False
"seven" in "Four score and seven years ago" # True
"wow" not in "That's impressive" # True
```

Looping with Strings

We can **loop over characters in a string** by visiting each index.

If the string is `s`, the string's first index is `0` and the last index is `len(s)-1`. Use `range(len(s))` to visit all possible indexes.

```
s = "Hello World"
for i in range(len(s)):
    print(i, s[i])
```

If you need to **solve a problem with strings** that requires doing something with every character, you can use a loop.

Algorithm: Remove spaces from a string.

```
s = "Wow! This is so exciting!"
result = ""
for i in range(len(s)):
    if s[i] != " ": # note the space between the quotes!
        result = result + s[i]
print(result) # "Wow!Thisissoexciting!"
```

We can use **string methods** to check if certain properties about strings is true.

A method is different from a built-in function, **it belongs to the object** and has the syntax `object.method()`.

```
a = s.lower() # a = "hello"

b = s.upper() # b = "HELLO"
```

`s.isdigit()`, `s.islower()`, and `s.isupper()` return `True` if the string is all-digits, all-lowercase, or all-uppercase, respectively.

```
s = "12345"
s.isdigit() # True
```

We will talk more about string methods next lecture!

Activity: Coding with Strings

You might be able to recognize a person by the types of punctuation they use in text messages. Maybe one friend loves exclamation points while another friend never uses them.

You do: write a function `getPunctuationFrequency(text, punc)` that takes a text message (a string) and a punctuation character (another string) and returns the frequency of how often that character appears in the text compared to other characters - the number of times it appears over the total number of characters.

For example, `getPunctuationFrequency("That's so exciting!! Good for you man!", "!")` would return ~0.079, because exclamation marks form 3/38 = ~0.079 as a ratio of the characters in the text.

On your own: Try it with real data!

We can try running our analysis function on real texts!

Websites like Project Gutenberg make the text of books available online for free. You can copy that text into a string, then run that string through the function.

Running the function through some popular classic fiction and trying out a few different types of punctuation already gleans interesting results. For example, the character . takes up 1.15% of text in *The Great Gatsby* compared to 0.82% in *Pride and Prejudice*; on the other hand, the character ; takes up only 0.03% of text in *The Great Gatsby* compared to 0.21% of text in *Pride and Prejudice*.

Combining these frequencies together can give us an interesting map of the writing styles of different authors!

Learning Goals

- **Index** and **slice** into strings to break them up into parts
- Use for loops to loop over strings by **index**

Sidebar: When do we use len(s) vs. len(s)-1?

It can be hard to tell when to use `len(s)` vs. `len(s)-1`. What do these two expressions really mean?

`len(s)` is the **length** of the string, the number of characters it contains. Because the first index of a string is 0, not 1, `s[len(s)]` returns an error.

On the other hand, `s[len(word):]` creates a slice that starts exactly `len(word)` characters into the string, which could be useful.

`len(s)-1` is the **last index** of the string. `s[len(s)-1]` returns the last character of a string.