CS 106AX Programming Methodologies in JavaScript and Python Autumn 2026



Lecture 15: Lists in Python



Written by Ben Yan, from materials by Jerry Cain

Lists Syntax

From even the old, medieval days of computing, when computers were dragons that filled entire rooms, programming languages have backed the notion of an ***array***: an **ordered sequence of values in memory**.

The individual values in an array are called ***elements***, and each element is identified by a unique position number called an ***index***, starting from `0` in Python and JavaScript, and extending up to the ***length*** of the array minus one. Here, the ***length*** of an array refers to its number of elements.

Python architects the array concept into a more general, flexible container called a ***list***. Though ***list*** and ***array*** are often used interchangeably in Python—which I'm very guilty of—we'll try to use the former ☺.

The simplest way to create a list in Python is to specify each of its elements, separated by commas, and surrounded by a hug of square brackets. For example, the declaration of `primes` below,

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

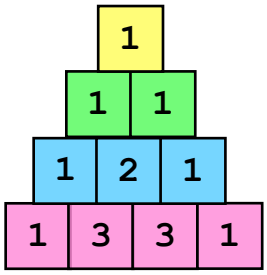
manufactures a list of the first ten prime numbers. Here are the indices of each element of `primes`.

element	2	3	5	7	11	13	17	19	23	29
index	0	1	2	3	4	5	6	7	8	9

As such, `primes[0]` refers to the `2`; `primes[9]` and `primes[-1]` are synonyms that refer to the `29`.

Python lists, like arrays in JavaScript, **can store elements of any type, including lists** (i.e. nested lists, or lists inside lists). The elements of a single array can be of differing type, though this is somewhat rare.

```
lectures = ["Monday", "Wednesday", "Friday"]
coins = [1, 5, 10, 25, 50, 100]
board = [["X", "X", "O"],
         [" ", "O", " "],  # each element of board is also a list!
         [" ", " ", " "]]
pascal = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]
```



Python List Patterns

The traditional `for` loop pattern—which we used to cycle over all characters in a string—generalizes to all Python lists, and sequences more broadly. The standard `for` loop to iterate over lists appears as this:

```
for element in list:
    lines of code (which perform some operations with the element)
```

Alternatively, if the index of each element is relevant to the operation we perform on it, we can write:

```
for i in range(len( list )):
      element = list [i]
      lines of code (which perform some operations with the index i and the element)
```

Note you use the built-in **`len`** function to obtain a list's length, as you do with strings. As an exemplar of the **`for`** loop pattern, the following function accepts a list of numbers as input, and returns their sum.

```
def sumIntegerList(list):
      sum = 0
      for number in list:
            sum += number
      return sum
```

As a second example, the **`superstring`** function accepts a list of **strings** as input, and returns a concatenated superstring consisting of each word and its index within the list. For instance, the call **`superstring(["red", "green", "blue"])`** returns the string **`"0red1green2blue"`**.

```
def superstring(wordList):  # my own example! not in the lecture slides ☺
      result = ""
      for index in range(len(wordList)):
            word = wordlist[index]
            result += str(index) + word
      return result
```

Python Sequences

The features listed above for ***lists*** are probably eerily reminiscent of those for ***strings***. If you're getting *déjà vu* (amazing Olivia Rodrigo song, by the way), your feeling is correct! In Python, ***strings*** and ***lists*** are both proud members of a more general class / constellation of objects called ***sequences***, and **all *sequences* support the following operations** ☺.



- *The* **`len`** *function*, which returns the length of the sequence, i.e. the number of elements / chars.
- *Index numbering beginning at* **`0`**, and extending up until the array's length minus one.
- *Negative index numbering* that counts backwards from the end, e.g., **`-k`** for the **`k`**th last element.
- *Selecting an individual element using square brackets*, e.g., **`seq[k]`** for the element at index **`k`**.
- *Slicing in all its colorful varieties*, e.g., supposing that list **`lst = [0, 10, 20, 30, 40]`**, then **`lst[0:3]`** creates the new sublist **`[0, 10, 20]`**, and **`lst[::2]`** creates **`[0, 20, 40]`**.
- *Concatenation via the* **`+`** *or* **`+=`** *operators*, e.g., **`[1,2] + [3,4]`** creates new list **`[1,2,3,4]`**.
- *Repetition using the* **`*`** *operator*, e.g., **`[0] * 4`** creates the list **`[0, 0, 0, 0]`**.
- *Inclusion testing using the* **`in`** *operator*, e.g., **`if "Ben" in nameList`**, or **`if ch in word.`**

One crucial difference between lists and strings, however, **is that the contents of a list can be changed in memory after creation**. For instance, ***list*`[k] = newValue`** is one way to reset an existing element.

Python List Methods

Below is a Louvre gallery of methods you can use to operate on strings. In practice, at least for CS106AX and JerryLand, you'll work mostly with ***append, insert, remove, pop, split, join***, ***index***, and ***reverse***.

Lists: Methods that Return Information
***list*`.index(`*value*`)`**
Returns the first index at which ***value*** appears in ***list***, or raises an error.
***list*`.index(`*value, start*`)`**
Returns the first index at which ***value*** appears in ***list*** after the ***start*** index***,*** or raises an error.
***list*`.count(`*value*`)`**
Returns the number of times ***value*** appears in ***list***.
***list*`.copy()`**
Creates and **returns** a brand new list whose elements are the same as the original.

To avoid a crashing error with the **`index`** function (or related methods), you should first check if the ***value*** appears in the ***list*** at all, e.g., **`if value in list`**, and only call **`index`** if this condition is true.

Lists: Methods that Add and Remove Elements
***list*`.append(`*value*`)`**
Adds ***value*** as a new element to the end of the list.
***list*`.insert(`*index*`, `*value*`)`**
Inserts ***value*** into the list so it now occupies the specified ***index***, shifting subsequent elements over.
***list*`.remove(`*value*`)`**
Removes the first instance of ***value***, or raises an error if it's not there.
***list*`.pop()`**

Removes and ***returns*** the last element of the list, e.g.,
`coins = [1, 5, 10, 25, 50, 100]`
`dollarCoin = coins.pop()`
Here, **`dollarCoin`** is the integer **`100`**, and **`coins`** is now **`[1, 5, 10, 25, 50]`**.


***list*`.pop(`*index*`)`**
Removes and ***returns*** the element at the specified ***index***, sliding down subsequent elements.
***list*`.clear()`**
Removes all elements from the list.

Lists: Methods that Reorder Elements In Place
***list*`.reverse()`**
Reverses the order of elements in the list in-place, i.e. does not create a new list, but modifies original.
***list*`.sort()`**
Sorts the elements of ***list*** in increasing order, modifying the original list in-place.
***list*`.sort(`*key*`)`**
Sorts the elements of ***list***, using the ***key*** function to generate a ranking value for each element.
***list*`.sort(`*key, reverse*`)`**
Sorts in descending order if the ***reverse*** flag is `True`.

To demonstrate how to use the `sort` function, take a close look at this example with a list of pupils.

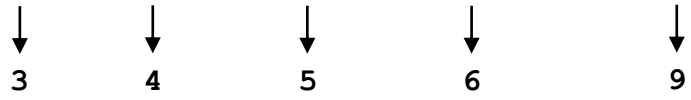
```
pupils = ["Bach", "Elizabeth", "David", "Amy", "Claire"]
pupils.sort()
```

The list would now be the alphabetically sorted `["Amy", "Bach", "Claire", "David", "Elizabeth"]`. However, **if we wanted to sort the strings by a different criterion, i.e. length**, we could pass in `len` as the ***key*** function, which generates the "ranking" value the `sort` function actually uses to sort each element. To perform a custom sort, you can write your own ***key*** function and pass it in!

```
words.sort(key = len)  # len is the built-in length function for strings!
```

And now, the array of words is `["Amy", "Bach", "David", "Claire", "Elizabeth"]`.

We sort elements by increasing `key(elem)`*, not the raw value of* `elem` *itself!*



Lists: Methods that Involve Strings
***str*`.split()`**
Splits a string into a list of its components using whitespace as separator, and **returns** that list, e.g., `"Mike Harvey Donna".split()` returns the list of names `["Mike", "Harvey", "Donna"]`.
***str*`.find(`*pattern, k*`)`**
Split a string into a list using the specified separator, and ***returns*** that list, e.g.,`"507-244-0751".split("-")` returns the list `["507", "244", "0751"]`. That's definitely not my telephone…
***str*`.splitlines()`**
Splits a string into separate lines at instances of the newline character `"\n"`, and **returns** that list.
***sep*`.join(`*list*`)`**
Joins the elements of ***list*** into a string, using ***sep*** as the separator, and **returns** that string, e.g., `"->".join(["Pichu","Pikachu","Raichu"])` returns `"Pichu->Pikachu->Raichu"`.

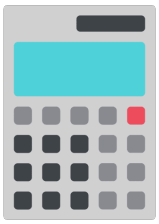
Generating Right-Truncatable Primes

The following exercise in Python and hobbyist mathematics relies on the notion of a ***right-truncatable prime***, which is **a prime that remains prime as you remove its digits from the right, one at a time**.

For instance, `7393` is right-truncatable, because `7393`, `739`, `73`, and `7` are all prime numbers ☺.

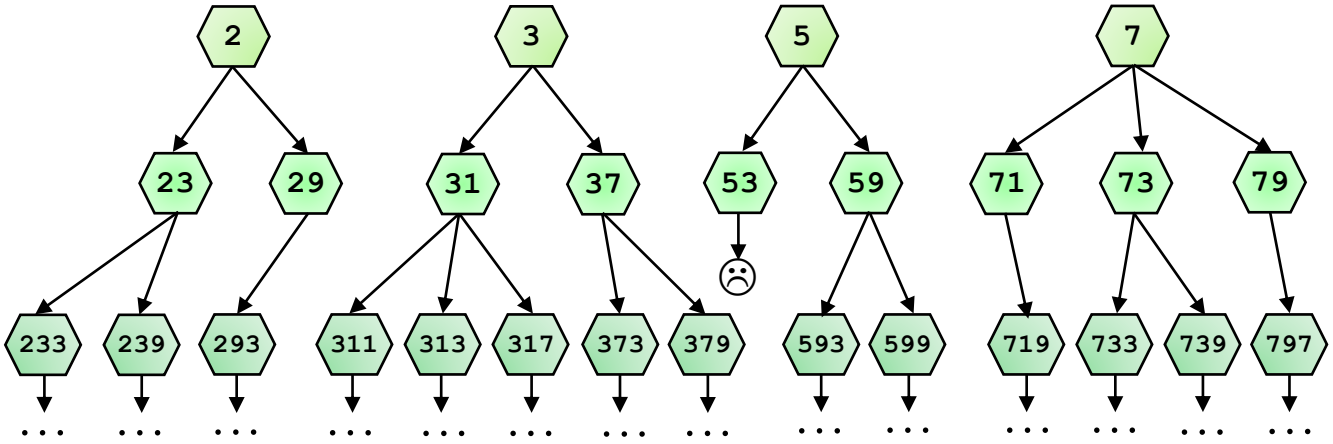


Prime? Yes ✓ *Prime?* Yes ✓ *Prime?* Yes ✓ *Prime?* Yes ✓



Every right-truncatable prime with `k` digits originates from appending a digit to the right end of one with `k - 1` digits. For instance, the two-digit prime `13` is right-truncatable. By appending a `1` and `7`, we generate `131` and `137`, which are both primes, so `131` and `137` are three-digit right-truncatable primes. That make sense?

Therefore, **you can generate all right-truncatable primes in rounds**, starting with single-digit `2, 3, 5, 7`. Then, you can try adding each of the digits `0` through `9` on every prime from the previous round, seeing if the result is prime—in which case, it's right-truncatable ☺.



🌴 Above is a tree diagram illustrating the generation process. Each layer of tree canopy corresponds to a single round, i.e. a list of right-truncatable primes with the same number of digits. **Here's a helper that performs a round of generation**, accepting the previous layer of primes as input to create the next layer.

```
def generateNextPrimes(currentPrimes):
      """ Generates list of k-digit right-truncatable primes from the list of (k – 1) digit such primes.
      """
      nextList = []
      for prime in currentPrimes:
            for digit in range(10):
                  candidate = 10 * prime + digit  # append new digit to right end
                  if isPrime(candidate):  # we'll implement this helper function next!
                        nextList.append(candidate)

      return nextList
```

The above relies on a helper that checks if **`n`** is **prime** or not—that is, **it has no divisors other than itself and `1`**. We handle **`2`** as a special case, and otherwise check only odd divisors up to $\sqrt{n}$ for speed ☺.

```
def isPrime(n):
      if n < 2:  # smallest prime is 2
            return False

      if n % 2 == 0:  # if n is even, the only way it's prime is if it equals 2
            return n == 2

      divisor = 3  # n is odd, so we only need to check for smaller odd divisors (3, 5, 7, 9, …)
      while divisor <= n ** 0.5:
            if n % divisor == 0:  # is divisible, so a composite number
                  return divisor
            divisor += 2

      return True
```

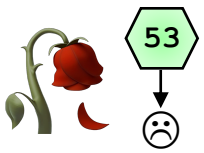
Lastly, we implement a function that generates all truncatable primes, starting from the top tree layer of **`[2, 3, 5, 7]`**, and using **`generateNextPrimes`** to repeatedly create each layer from the previous.

```
def generateTruncatablePrimes():
      """ Returns a list whose element at index k is a list of all the k-digit right-truncatable primes,
          in increasing order ☺, i.e. [[], [2, 3, 5, 7], [23, 29, 31, 37, ...], ...]
      """
      primes = []
      primes.append([])  # 0-digit primes, included so that index matches with number of digits
      primes.append([2, 3, 5, 7])  # 1-digit primes

      while True:  # keeps going until we can't generate any larger primes
            prevList = primes[-1]
            nextList = generateNextPrimes(prevList)

            if len(nextList) == 0:  # no more primes in the next layer, so tree has died out
                  return primes

            primes.append(nextList)
```

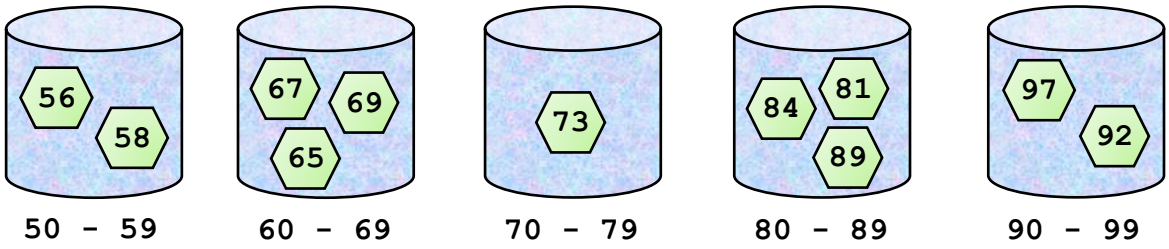


Indeed, **there are finitely many right-truncatable primes**—hence, the program above doesn't cause an infinite loop ☺. The largest such prime is the 10-digit colossal titan **`73,939,133`**, and there are precisely 83 (!!) of these 10-digit truncatable primes. Sadly, **there is no way of extending any of these 10-digit primes into an 11-digit prime** (I won't make you do the painful math), so all branches here—with no more room to grow—wilt away and die.

Maybe you're curious what really happens in the code when the generation ends. Well, in that final, fateful iteration, **`prevList`** is the list of 10-digit truncatable primes, but **`nextList`** (with no possible larger primes) stays empty, so the loop quietly accepts its mortality, and ***returns*** the final **`primes`** list 🪦.

Count Letter Frequencies

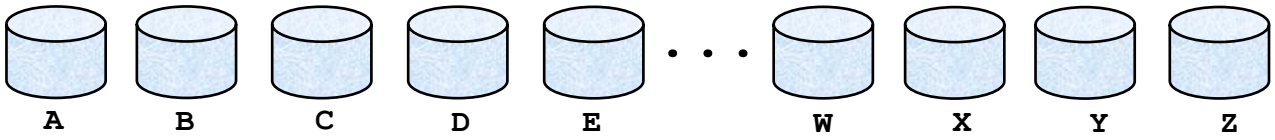
Lists are often very useful when you have a messy (though delicious and fragrant) soup of data values, and **need to count how many values fall into each bucket of ranges**, a process called ***tabulation***.



An example of tabulation is a program that counts **how many times each of the 26 letters appears in a body of text**. Such a program is useful for performing ***frequency analysis***, a shimmeringly powerful tool to decrypt and demystify letter-substitution ciphers—with the notion that *the most frequently appearing letters in the ciphertext likely correspond to the most frequent letters of the alphabet*, e.g., **`E`**, **`T`**, **`A`**, **`O`**, **`I`**.

First, we implement the function **`createFrequencyTable`**, which **manufactures and returns a list of 26 elements indicating the counts for each alphabet letter**. All counts are initialized to **`0`** pre-tabulation.

```
def createFrequencyTable():
    return [ 0 ] * 26      # clever way to create a list of 26 zeroes ☺
```



Next, we read in a source string, parse through its letters, and update the counters above. **But given a letter `ch`, how do we know its counter index?** (e.g., **`"A"`** has index 0, **`"B"`** has index 1, and so on)

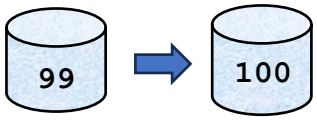
One tidbit you'll need to know is that Python includes **two built-in functions**, **`chr`** and **`ord`**, **to facilitate conversion between an integer, and the corresponding Unicode character**. If this seems familiar, you really paid attention during our JavaScript lectures, kudos to you! They are Python's counterparts to the **`String.fromCharCode`** and ***`str`*`.charCodeAt`** methods you've seen in JavaScript, respectively ☺.

`chr(`*`code`*`)`	Accepts as an integer ***code*** as input, and **returns** a one-character string containing the Unicode character assigned that ***code***. As colorful examples, ○ **`chr(32) -> " "`** (space character) ○ **`chr(67) -> "C"`** (an uppercase C) ○ **`chr(960) -> "π"`** (the Greek letter and famous geometric constant)
`ord(`*`char`*`)`	Accepts a one-character string ***char*** as input, and returns the value of that character in Unicode. As scintillant examples, ○ **`ord(" ") -> 32`** ○ **`chr("C") -> 67`** ○ **`chr("π") -> 960`**

For a letter **ch**, we can compute its index by converting to uppercase, and taking the difference between its Unicode value and that of **"A"**. For instance, **ord("B") – ord("A")** gives **1**, **ord("C") – ord("A")** gives **2**, and so on, until **ord("Z") – ord("A")**, which gives **25**. How cool is that!? ☺

As such, the following method scans over the characters of a new string—and if a given character is alphabetic, **"A", "B", ..., "Z", "a", "b", ..., "z"**, we compute its index in the frequency table **counts**, and increment that counter. **Note the counters are not case-sensitive**, e.g., **counts[0]** records the number of **"A"** or **"a"** letters we've encountered so far (both uppercase and lowercase).

```
def updateFrequencyTable(counts, source):
      for ch in source:
            if ch.isalpha():
                  index = ord(ch.upper()) - ord("A")
                  counts[index] += 1
```



Next, we write a logging function that, given the current frequency data stored in the **count** list, **prints out each of the 26 letters, and how many times each one has appeared so far**. However, letters whose count is **0**—that is, we haven't encountered them just yet—are not included in the display.

```
def printFrequencyTable(counts):
      for i in range(len(counts)):
            count = counts[i]

            if count > 0:
                  ch = chr(ord("A") + i)
                  print(ch + ": " + str(count))
```

Note the expression **ord("A") + i** computes the Unicode value of the **i**th letter of the alphabet, and the **chr()** function call converts it back into its letter form. An example print out might appear like:

```
A: 40
B: 20
... more entries omitted for conciseness ...
Z: 5
```

Lastly, we wrap everything in an interactive console program! The user is repeatedly prompted to enter lines, and they indicate they're all done by entering a blank line. **We keep a table of letter counters, and for each line entered in the loop, we update the table**. At the end, we print out each letter's count.

```
      def CountLetterFrequencies():
            counts = createFrequencyTable()
            print("Enter input lines, ending with a blank line.")

            while True:
                  line = input("> ")
                  if line == "": break
                  updateFrequencyTable(counts, line)

          printFrequencyTable(counts)
```

A fun sample run of the program glitters the snowy top and summit of the next page.

```
Terminal: CountLetterFrequencies
Enter input lines, ending with a blank line.

> Nice to meet you, where you been?
> I could show you incredible things
> Magic, madness, heaven, sin
> Saw you there, and I thought
>

A: 5
B: 2
C: 4
D: 4
E: 14
G: 3
H: 7
I: 8
L: 2
M: 3
N: 8
O: 8
R: 3
S: 6
T: 6
U: 6
V: 1
W: 3
Y: 4
```

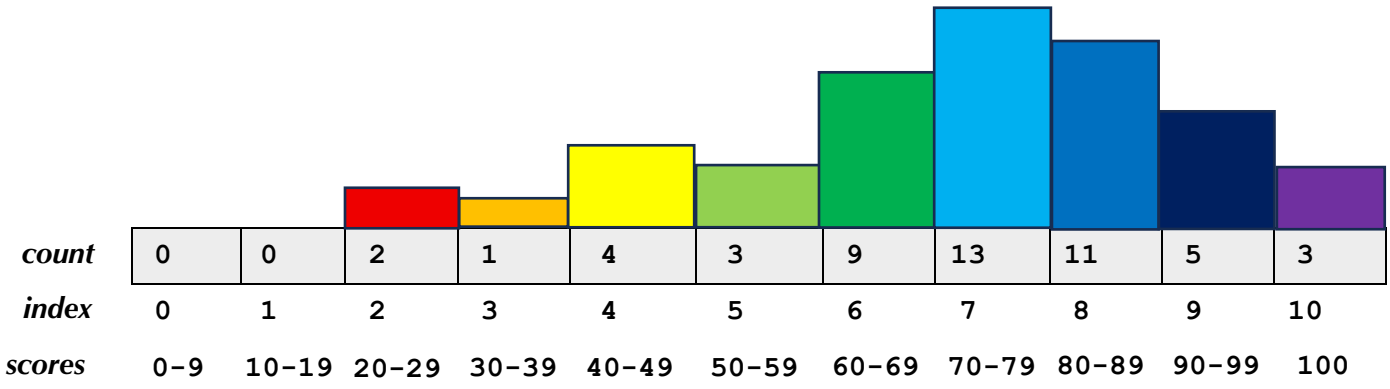


Note that the letters **F, J, K, P, Q, X,** and **Z** are omitted, since they don't appear in the inputted text ☹.

Histogram

We'll write a function, **`displayHistogram(scores)`**, which takes a list of scores in the range **`0`** to **`100`**, then displays on the console a histogram of those scores partitioned into ranges **`0`** to **`9`**, **`10`** to **`19`**, and so on, up to the range **`90`** to **`99`**. We reserve a special final range for only ⭐ perfect **`100`** scores ⭐.

Thus, our **`histogram`** is a Python list of **`11`** counters, one per range. An example is illustrated below.

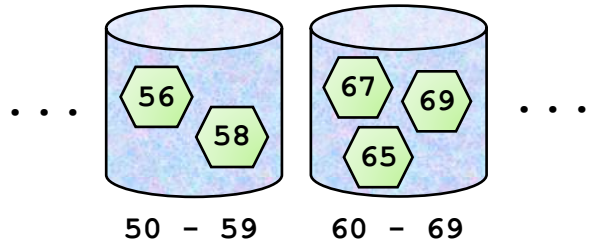


Here's an implementation of the helper function **`createHistogram`**, which builds the **`histogram`** list of counters from an input list of **`scores`**. No printing or console-based activity is involved just yet.

```
def createHistogram(scores):
    histogram = [ 0 ] * 11

    for score in scores:
        if score >= 0 and score <= 100:
            range = score // 10
            histogram[range] += 1

    return histogram
```



Note that **`score // 10`** uses integer division to give the range to which that **`score`** belongs. For instance, in my advanced algorithms class last year, I received an utterly heart-breaking grade of **`52`** on the final 💔, which belongs to range **`50 - 59`** or the counter **`histogram[5]`**, since **`52 // 10 => 5`**.

Lastly, given a fully constructed **`histogram`** list, we print out each range in the console, and how many scores belong to that range. Though histograms are canonically presented vertically (i.e. with skyscraper bars), it's much easier to print horizontally with code, and as such, that's how this histogram is drawn ☺

```
def displayHistogram(histogram):
    for i in range(len(histogram)):
        if i == 0:  # special case for 0 – 9 range (otherwise we'd display 0s, not 00s)
            label = "00s: "
        elif i == 10:  # special case for perfect 100 scores
            label = "100: "
        else:
            label = str(10 * i) + "s: "

        print(label + "*" * histogram[i])  # prints one star per score in this range
```

For the colorful histogram of scores on the previous page, here's how our program's output would look.

Terminal: Histogram

```
00s:
10s:
20s: **
30s: *
40s: ****
50s: ***
60s: *********
70s: ************
80s: ***********
90s: *****
100: ***
```



Translation from Japanese:
You did great!!