Python
Lecture 6

Strings

Chapter 1

Introduction to Strings

Introduction to Strings

Words and sentences are fundamental to how we communicate, so it follows that we'd want our computers to be able to work with words and sentences as well.

In Python, the way we store something like a word, a sentence, or even a whole paragraph is as a **string**. A string is a sequence of characters contained within a pair of `'single quotes'` or `"double quotes"`. A string can be any length and can contain any letters, numbers, symbols, and spaces.

In this lesson, we will learn more about strings and how they are treated in Python. We will learn how to slice strings, select specific characters from strings, search strings for characters, iterate through strings, and use strings in conditional statements.

Let's get started.

Example

```
favorite_word = "Ehen"
print(favorite_word)
```

They're all Lists!

A string can be thought of as a **list** of characters.

Like any other list, each character in a string has an index. Consider the string:

```
favorite_fruit = "blueberry"
```

We can select specific letters from this string using the *index*. Let's look at the first letter of the string.

```
print(favorite_fruit[1])
# Output: l
```

Whoops, is that the first letter you expected? Notice that the letter at index `1` of `"blueberry"` isn't `b`, it's `l`. This is because the indices of a string start at `0`. `b` is located at the zero index and we could select it using:

```
print(favorite_fruit[0])
# Output: b
```

It's important to note that indices of strings must be integers. If we were to try to select a non-integer index we would get a `TypeError`.

For example:

```
print(favorite_fruit[1.5])
```

This would result in:

```
Traceback (most recent call last):
  File "script.py", line 3, in <module>
    print(favorite_fruit[1.5])
TypeError: string indices must be integers
```

Example

```
my_name = "Joseph"
first_initial = my_name[0]
```

Cut Me a Slice of String

Not only can we select a single character from a string, but we can also select entire chunks of characters from a string. We can do this with the following syntax:

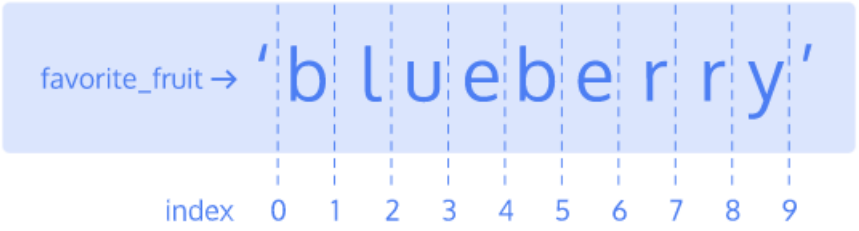
```
string[first_index:last_index]
```

This is called *slicing* a string. When we slice a string we are creating a brand new string that starts at (and includes) the `first_index` and ends at (but excludes) the `last_index`.

Let's look at some examples of this. Recall our favorite fruit:

```
favorite_fruit = "blueberry"
```

The indices of this string are shown in the diagram below.



Let's say we wanted a new string that contains the letters `be`. We could slice `favorite_fruit` as follows:

```
print(favorite_fruit[4:6])
# Output: be
```

Notice how the character at the first index, `b`, is included, but the character at the last index, `r`, is excluded. If you look for the indices 4 and 6 in the diagram, you can see how the `r` is outside that range.

We can also have open-ended selections. If we remove the first index, the slice starts at the beginning of the string and if we remove the second index the slice continues to the end of the string.

```
print(favorite_fruit[:4])
# Output: blue

print (favorite_fruit[4:])
# Output: berry
```

Again, notice how the `b` from `berry` is excluded from the first example and included in the second example.

Example

```
first_name = "Rodrigo"
last_name = "Villanueva"

new_account = last_name[:5]
temp_password = last_name[2:6]
```

Concatenating Strings

We can also *concatenate*, or combine, two existing strings together into a new string. Consider the following two strings:

```
fruit_prefix = "blue"
fruit_suffix = "berries"
```

We can create a new string by concatenating them together as follows:

```
favorite_fruit = fruit_prefix + fruit_suffix

print(favorite_fruit)
# Output: blueberries
```

Notice that there are no spaces added here. We have to manually add in the spaces when concatenating strings if we want to include them.

```
fruit_sentence = "My favorite fruit is" + favorite_fruit

print(fruit_sentence)
# Output: My favorite fruit isblueberries
```

```
fruit_sentence = "My favorite fruit is " + favorite_fruit

print(fruit_sentence)
# Output: My favorite fruit is blueberries
```

It's subtle, but notice that in the first example, there is no space between "is" and "blueberries".

Example

```
first_name = "Julie"
last_name = "Blevins"
def account_generator(x, y):
  x = first_name[:3]
  y = last_name[:3]
  account_name = x + y
  return account_name

new_account = account_generator(first_name, last_name)
```

More and More String Slicing (How Long is that String?)

Python comes with some built-in functions for working with strings. One of the most commonly used of these functions is `len()`. `len()` returns the number of characters in a string:

```
favorite_fruit = "blueberry"

length = len(favorite_fruit)

print(length)
# Output: 9
```

If you are taking the length of a sentence the spaces are counted as well.

```
fruit_sentence = "I love blueberries"

print(len(fruit_sentence))
# Output: 18
```

`len()` comes in handy when we are trying to select characters from the end of a string. What is the index of the last character,`"y"`, of `favorite_fruit` from above? You can try to run the following code:

```
last_char = favorite_fruit[len(favorite_fruit)]

print(last_char)
```
This will result in:

```
IndexError: string index out of range
```
Why does this generate an `IndexError`? Because the indices start at `0`, so the final character in `favorite_fruit` has an index of `8`. `len(favorite_fruit)` returns `9` and, because there is no value at that index, an `IndexError` occurs.

Instead, the last character in a string has an index that is `len(string_name) - 1`.

```
last_char = favorite_fruit[len(favorite_fruit)-1]

print(last_char)
# Output: y
```
You could also slice the last several characters of a string using `len()`:

```
length = len(favorite_fruit)
last_chars = favorite_fruit[length-4:]
print(last_chars)
# Output: erry
```
Using a `len()` statement as the starting index and omitting the final index lets you slice `n` characters from the end of a string, where `n` is the amount you subtract from `len()`.

Example

```
first_name = "Reiko"
last_name = "Matsuki"

def password_generator(x, y):
  xlength = len(first_name)
  ylength = len(last_name)
  x = first_name[xlength-3:]
  y = last_name[ylength-3:]
  account_name = x + y
  return account_name

temp_password = password_generator(first_name, last_name)
```

Negative Indices
In the previous exercise, we used `len()` to get a slice of characters at the end of a string.

There's a much easier way to do this — we can use negative indices! Negative indices count backward from the end of the string, so `string_name[-1]` is the last character of the string, `string_name[-2]` is the second last character of the string, etc.

Here are some examples:

```
favorite_fruit = 'blueberry'
print(favorite_fruit[-1])
# => 'y'

print(favorite_fruit[-2])
# => 'r'

print(favorite_fruit[-3:])
# => 'rry'
```

Notice that we are able to slice the last three characters of 'blueberry' by having a starting index of `-3` and omitting a final index.

Example

```
company_motto = "Copeland's Corporate Company helps you capably cope with the constant cacophony of daily life"
second_to_last = company_motto[-2]
final_word = company_motto[-4:]
```

Strings are Immutable
So far in this lesson, we've been selecting characters from strings, slicing strings, and concatenating strings. Each time we perform one of these operations we are creating an entirely new string.

This is because strings are *immutable*. This means that we cannot change a string once it is created. We can use it to create other strings, but we cannot change the string itself.

This property, generally, is known as *mutability*. Data types that are *mutable* can be changed, and data types, like strings, that are *immutable* cannot be changed.

Example

```
first_name = "Bob"
last_name = "Daily"

fixed_first_name = "R" + first_name[1:]
```

Escape Characters

Occasionally when working with strings, you'll find that you want to include characters that already have a special meaning in python. For example let's say I create the string

```
 favorite_fruit_conversation = "He said, "blueberries are my favorite!""
```

We'll have accidentally ended the string before we wanted to by including the `"` character. The way we can do this is by introducing *escape characters*. By adding a backslash in front of the special character we want to escape, `\"`, we can include it in a string.

```
favorite_fruit_conversation = "He said, \"blueberries are my favorite!\""
```

Now it works!

Example

```
password = "theycallme\"crazy\"91"
```

Iterating through Strings

Now you know enough about strings that we can start doing the really fun stuff!

Because strings are lists, that means we can iterate through a string using `for` or `while` loops. This opens up a whole range of possibilities of ways we can manipulate and analyze strings. Let's take a look at an example.

```
def print_each_letter(word):
  for letter in word:
    print(letter)
```

This code will iterate through each letter in a given word and will print it to the terminal.

```
favorite_color = "blue"
print_each_letter(favorite_color)
```

```
# => 'b'
# => 'l'
# => 'u'
# => 'e'
```

Let's try a couple of problems where we need to iterate through a string.

Example

```
def get_length(word):
  count = 0
  for x in word:
    count += 1
  return count

get_length("Hurray!!!")
```

Strings and Conditionals (Part One)

Now that we are iterating through strings, we can really explore the potential of strings. When we iterate through a string we do *something* with each character. By including conditional statements inside of these iterations, we can start to do some really cool stuff.

Take a look at the following code:

```
favorite_fruit = "blueberry"
counter = 0
for character in favorite_fruit:
  if character == "b":
    counter = counter + 1
print(counter)
```

This code will count the number of `b`s in the string "blueberry" (hint: it's two). Let's take a moment and break down what exactly this code is doing.

First, we define our string, `favorite_fruit`, and a variable called `counter`, which we set equal to zero. Then the `for` loop will iterate through each character in `favorite_fruit` and compare it to the letter `b`.

Each time a character equals `b` the code will increase the variable `counter` by one. Then, once all characters have been checked, the code will print the counter, telling us how many `b`s were in "blueberry". This is a great example of how iterating through a string can be used to solve a specific application, in this case counting a certain letter in a word.

Example

```
def letter_check(word, letter):
  if letter in word:
    return True
  else:
    return False

letter_in_word = letter_check("Joseph", "q")
print(letter_in_word)
```

Strings and Conditionals (Part Two)

There's an even easier way than iterating through the entire string to determine if a character is in a string. We can do this type of check more efficiently using `in`. `in` checks if one string is part of another string.

Here is what the syntax of `in` looks like:

```
letter in word
```

Here, `letter in word` is a boolean expression that is `True` if the string `letter` is in the string `word`. Here are some examples:

```
print("e" in "blueberry")
# => True
print("a" in "blueberry")
# => False
```

In fact, this method is more powerful than the function you wrote in the last exercise because it not only works with letters, but with entire strings as well.

```
print("blue" in "blueberry")
# => True
print("blue" in "strawberry")
# => False
```

Example

1. Write a function called `contains` that takes two arguments, `big_string` and `little_string` and returns `True` if `big_string` contains `little_string`.

For example `contains("watermelon", "melon")` should return `True` and `contains("watermelon", "berry")` should return `False`.

```
def contains(big_string, little_string):
  return little_string in big_string

print(contains('watermelon', 'melon'))
```

2. Write a function called `common_letters` that takes two arguments, `string_one` and `string_two` and then returns a list with all of the letters they have in common.

```
def common_letters(string_one, string_two):
  lst = []
  for x in string_one:
    if (x in string_two) and  (x not in lst):
      lst.append(x)
  return lst

print(common_letters('python', 'ruby on rails'))
```

Chapter 2

Strings Methods

Introduction

Do you have a gigantic string that you need to parse for information? Do you need to sanitize a users input to work in a function? Do you need to be able to generate outputs with variable values? All of these things can be accomplished with *string methods*!

Python comes with built-in *string methods* that gives you the power to perform complicated tasks on strings very quickly and efficiently. These string methods allow you to change the case of a string, split a string into many smaller strings, join many small strings together into a larger string, and allow you to neatly combine changing variables with string outputs.

In the previous lesson, you worked with `len()`, which was a *function* that determined the number of characters in a string. This, while similar, was NOT a string method. String methods all have the same syntax:

```
string_name.string_method(arguments)
```

Unlike `len()`, which is called with a string as its argument, a string method is called at the end of a string and each one has its own method specific arguments.

Formatting Methods

There are three string methods that can change the casing of a string. These are `.lower()`, `.upper()`, and `.title()`.

- `.lower()` returns the string with all lowercase characters.
- `.upper()` returns the string with all uppercase characters.
- `.title()` returns the string in title case, which means the first letter of each word is capitalized.

Here's an example of `.lower()` in action:

```
favorite_song = 'SmOoTH'
favorite_song_lowercase = favorite_song.lower()
print(favorite_song_lowercase)
# => 'smooth'
```

Every character was changed to lowercase! It's important to remember that string methods can only **create** new strings, they do not change the original string.

```
print(favorite_song)
# => 'SmOoTH'
```

See, it's still the same! These string methods are great for sanitizing user input and standardizing the formatting of your strings.

Example

```
poem_title = "spring storm"
poem_author = "William Carlos Williams"

poem_title_fixed = poem_title.title()
print(poem_title)
print(poem_title_fixed)

poem_author_fixed = poem_author.upper()
print(poem_author)
print(poem_author_fixed)
```

Splitting Strings

`.upper()`, `.lower()`, and `.title()` all are performed on an existing string and produce a string in return. Let's take a look at a string method that returns a different object entirely!

`.split()` is performed on a string, takes one argument, and returns a list of substrings found between the given argument (which in the case of `.split()` is known as the delimiter). The following syntax should be used:

```
string_name.split(delimiter)
```

If you do not provide an argument for `.split()` it will default to splitting at spaces.

For example, consider the following strings:

```
man_its_a_hot_one = "Like seven inches from the midday sun"
print(man_its_a_hot_one.split())
# => ['Like', 'seven', 'inches', 'from', 'the', 'midday', 'sun']
```

`.split` returned a list with each word in the string. Important to note: if we run `.split()` on a string with no spaces, we will get the same string in return.

Example

```
line_one = "The sky has given over"
line_one_words = line_one.split()
```

Splitting Strings II

If we provide an argument for `.split()` we can dictate the character we want our string to be split on. This argument should be provided as a string itself.

Consider the following example:

```
greatest_guitarist = "santana"
print(greatest_guitarist.split('n'))
# => ['sa', 'ta', 'a']
```

We provided `'n'` as the argument for `.split()` so our string "santana" got split at each `'n'` character into a list of three strings.

What do you think happens if we split the same string at `'a'`?

```
print(greatest_guitarist.split('a'))
# => ['s', 'nt', 'n', ' ']
```

Notice that there is an unexpected extra `''` string in this list. When you split a string on a character that it also ends with, you'll end up with an empty string at the end of the list.

You can use *any* string as the argument for `.split()`, making it a versatile and powerful tool.

Example

```
authors = "Audre Lorde,Gabriela Mistral,Jean Toomer,An Qi,Walt Whitman,Shel Si
lverstein,Carmen Boullosa,Kamala Suraiyya,Langston Hughes,Adrienne Rich,Nikki
Giovanni"

author_names = authors.split(',')
author_last_names = []
for x in author_names:
  split_names = x.split(" ")
  author_last_names.append(split_names[-1])
print(author_last_names)
```

Splitting Strings III

We can also split strings using *escape sequences*. Escape sequences are used to indicate that we want to split by something in a string that is not necessarily a character. The two escape sequences we will cover here are

- `\n` Newline
- `\t` Horizontal Tab

Newline or `\n` will allow us to split a multi-line string by line breaks and `\t` will allow us to split a string by tabs. `\t` is particularly useful when dealing with certain datasets because it is not uncommon for data points to be separated by tabs.

Let's take a look at an example of splitting by an escape sequence:

```
smooth_chorus = \
"""And if you said, "This life ain't good enough."
I would give my world to lift you up
I could change my life to better suit your mood
Because you're so smooth"""

chorus_lines = smooth_chorus.split('\n')

print(chorus_lines)
```

This code is splitting the multi-line string at the newlines (`\n`) which exist at the end of each line and saving it to a new list called `chorus_lines`. Then it prints `chorus_lines` which will produce the output

```
['And if you said, "This life ain\'t good enough."', 'I would give
my world to lift you up', 'I could change my life to better suit
your mood', "Because you're so smooth"]
```

The new list contains each line of the original string as its own smaller string. Also, notice that Python automatically escaped the `'` character in the first line and adjusted to double quotation marks to allow the apostrophe on last line when it created the new list.

Example

```
spring_storm_text = \
"""The sky has given over
its bitterness.
Out of the dark change
all day long
rain falls and falls
as if it would never end.
Still the snow keeps
its hold on the ground.
But water, water
from a thousand runnels!
It collects swiftly,
```

```
dappled with black
cuts a way for itself
through green ice in the gutters.
Drop after drop it falls
from the withered grass-stems
of the overhanging embankment."""

spring_storm_lines = spring_storm_text.split('\n')
```

Joining Strings

Now that you've learned to break strings apart using `.split()`, let's learn to put them back together using `.join()`. `.join()` is essentially the opposite of `.split()`, it *joins* a list of strings together with a given delimiter. The syntax of `.join()` is:

```
'delimiter'.join(list_you_want_to_join)
```

Now this may seem a little weird, because with `.split()` the argument was the delimiter, but now the argument is the list. This is because *join* is still a string method, which means it has to act on a string. The string `.join()` acts on is the delimiter you want to join with, therefore the list you want to join has to be the argument.

This can be a bit confusing, so let's take a look at an example.

```
my_munequita = ['My', 'Spanish', 'Harlem', 'Mona', 'Lisa']
print(' '.join(my_munequita))
# => 'My Spanish Harlem Mona Lisa'
```

We take the list of strings, `my_munequita`, and we joined it together with our delimiter, `' '`, which is a space. The space is important if you are trying to build a sentence from words, otherwise, we would have ended up with:

```
print(''.join(my_munequita))
# => 'MySpanishHarlemMonaLisa'
```

Example

```
reapers_line_one_words = ["Black", "reapers", "with", "the", "sound", "of", "s
teel", "on", "stones"]

reapers_line_one = " ".join(reapers_line_one_words)
print(reapers_line_one)
```

Joining Strings II

In the last exercise, we joined together a list of words using a space as the delimiter to create a sentence. In fact, you can use any string as a delimiter to join together a list of strings. For example, if we have the list

```
santana_songs = ['Oye Como Va', 'Smooth', 'Black Magic Woman',
'Samba Pa Ti', 'Maria Maria']
```

We could join this list together with ANY string. One often used string is a comma `,` because then we can create a string of *comma separated variables*, or CSV.

```
santana_songs_csv = ','.join(santana_songs)
print(santana_songs_csv)
# => 'Oye Como Va,Smooth,Black Magic Woman,Samba Pa Ti,Maria Maria'
```

You'll often find data stored in CSVs because it is an efficient, simple file type used by popular programs like Excel or Google Spreadsheets.

You can also join using *escape sequences* as the delimiter. Consider the following example:

```
smooth_fifth_verse_lines = ['Well I\'m from the barrio', 'You hear
my rhythm on your radio', 'You feel the turning of the world so soft
and slow', 'Turning you \'round and \'round']

smooth_fifth_verse = '\n'.join(smooth_fifth_verse_lines)

print(smooth_fifth_verse)
```

This code is taking the list of strings and joining them using a newline `\n` as the delimiter. Then it prints the result and produces the output:

```
Well I'm from the barrio
You hear my rhythm on your radio
You feel the turning of the world so soft and slow
Turning you 'round and 'round
```

Example

```
winter_trees_lines = ['All the complicated details', 'of the attiring and', 't
he disattiring are completed!', 'A liquid moon', 'moves gently among', 'the lo
ng branches.', 'Thus having prepared their buds', 'against a sure winter', 'th
e wise trees', 'stand sleeping in the cold.']

winter_trees_full = '\n'.join(winter_trees_lines)
print(winter_trees_full)
```

.strip()
When working with strings that come from real data, you will often find that the strings aren't super clean. You'll find lots of extra whitespace, unnecessary linebreaks, and rogue tabs.

Python provides a great method for cleaning strings: `.strip()`. Stripping a string removes all whitespace characters from the beginning and end. Consider the following example:

```
featuring = "            rob thomas                    "
print(featuring.strip())
# => 'rob thomas'
```

All the whitespace on either side of the string has been stripped, but the whitespace in the middle has been preserved.

You can also use `.strip()` with a character argument, which will strip that character from either end of the string.

```
featuring = "!!!rob thomas       !!!!!"
print(featuring.strip('!'))
# => 'rob thomas       '
```

By including the argument `'!'` we are able to strip all of the `!` characters from either side of the string. Notice that now that we've included an argument we are no longer stripping whitespace, we are ONLY stripping the argument.

Example

```
love_maybe_lines = ['Always    ', '     in the middle of our bloodiest battles
  ', 'you lay down your arms', '           like flowering mines    ','\n' ,'
 to conquer me home.    ']

love_maybe_lines_stripped = [x.strip() for x in love_maybe_lines]

love_maybe_full = '\n'.join(love_maybe_lines_stripped)
print(love_maybe_full)
```

Replace
The next string method we will cover is `.replace()`. Replace takes two arguments and replaces all instances of the first argument in a string with the second argument. The syntax is as follows

```
string_name.replace(character_being_replaced, new_character)
```

Great! Let's put it in context and look at an example.

```
with_spaces = "You got the kind of loving that can be so smooth"
with_underscores = with_spaces.replace(' ', '_')
print(with_underscores)
# 'You_got_the_kind_of_loving_that_can_be_so_smooth'
```

Here we used `.replace()` to change every instance of a space in the string above to be an underscore instead.

Example

```
toomer_bio = \
"""
Nathan Pinchback Tomer, who adopted the name Jean Tomer early in his literary career, was born in Washington, D.C. in 1894. Jean is the son of Nathan Tomer was a mixed-
race freedman, born into slavery in 1839 in Chatham County, North Carolina. Je an Tomer is most well known for his first book Cane, which vividly portrays th e life of African-Americans in southern farmlands.
"""
toomer_bio_fixed = toomer_bio.replace("Tomer","Toomer")
print(toomer_bio_fixed)
```

.find()

Another interesting string method is `.find()`. `.find()` takes a string as an argument and searching the string it was run on for that string. It then returns the first *index value* where that string is located.

Here's an example:

```
print('smooth'.find('t'))
# => '4'
```

We searched the string `'smooth'` for the string `'t'` and found that it was at the fourth index spot, so `.find()` returned `4`.

You can also search for larger strings, and `.find()` will return the index value of the first character of that string.

```
print("smooth".find('oo'))
# => '2'
```

Notice here that `2` is the index of the *first* o.

Example

```
god_wills_it_line_one = "The very earth will disown you"
disown_placement = god_wills_it_line_one.find('disown')
```

.format()

Python also provides a handy string method for including variables in strings. This method is `.format()`. `.format()` takes variables as an argument and includes them in the string that it is run on. You include `{}` marks as placeholders for where those variables will be imported.

Consider the following function:

```
def favorite_song_statement(song, artist):
  return "My favorite song is {} by {}.".format(song, artist)
```

The function `favorite_song_statement` takes two arguments, song and artist, then returns a string that includes both of the arguments and prints a sentence.
Note: `.format()` can take as many arguments as there are `{}` in the string it is run on, which in this case is two.

Here's an example of the function being run:

```
print(favorite_song_statement("Smooth", "Santana"))
# => "My favorite song is Smooth by Santana"
```

Now you may be asking yourself, I could have written this function using string concatenation instead of `.format()`, why is this method better? The answer is legibility and reusability. It is much easier to picture the end result `.format()` than it is to picture the end result of string concatenation and legibility is everything. You can also reuse the same base string with different variables, allowing you to cut down on unnecessary, hard to interpret code.

Example

```
def poem_title_card(title, poet):
  return "The poem \'{}\' is written by {}.".format(title, poet)

print(poem_title_card("I Hear America Singing", "Walt Whitman"))
```

.format() II

`.format()` can be made even more legible for other people reading your code by including *keywords*. Previously, with `.format()`, you had to make sure that your

variables appeared as arguments in the same order that you wanted them to appear in the string, which added unnecessary complications when writing code.

By including keywords in the string, and in the arguments, you can remove that ambiguity. Let's look at an example.

```
def favorite_song_statement(song, artist):
    return "My favorite song is {song} by
{artist}.".format(song=song, artist=artist)
```

Now it is clear to anyone reading the string what it is supposed to return, they don't even need to look at the arguments of `.format()` in order to get a clear understanding of what is supposed to happen. You can even reverse the order of `artist` and `song` in the code above and it will work the same way. This makes writing AND reading the code much easier.

Example

```
def poem_description(publishing_date, author, title, original_work):
  poem_desc = "The poem {title} by {author} was originally published in {origi
nal_work} in {publishing_date}.".format(publishing_date=publishing_date, autho
r=author, title=title, original_work=original_work)
  return poem_desc

my_beard_description = poem_description(
  "1974",
  "Shel Silverstein",
  "My Beard",
  "Where the Sidewalk Ends"
)
print(my_beard_description)
```