

Lecture 14: Strings in Python



Written by Ben Yan, from materials by Jerry Cain

Strings and Indexing Syntax

Like JavaScript, **a string in Python is an ordered sequence of characters**. The position of each character, as in all reasonable programming languages, is an index ranging from `0` to the string's length minus `1`.

The code below initializes the famous *Hello World* string. Note that we don't use `let` keyword anymore to define variables ☺. Bye bye, sayonara JavaScript.




```
s = "Hello World!"
```

The characters and their corresponding indices are arranged as follows.

H	e	l	l	o		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

You can isolate an individual character using the syntax `str[k]`, where `k` is the index of the desired character. For instance, `s[6]` for the above string returns the one-character string `"W"`. **If `k` is out of bounds for string `str`**, i.e. greater than `len(k) - 1`, **Python will crash the program with an error**—unlike in JavaScript, where it would quietly evaluate to the value `undefined` without crashing.

What happens if the supplied index is negative then? (e.g., `-1`, `-2`, et cetera) Well, glad you asked ☺.

Negative Indexing

Python is quirky for many reasons (you'll see 😈), but one awesome quirk is that **you can also use negative indices to specify character positions, by counting backward from the end of the string**. The last character has index `-1`, the 2nd to last character has index `-2`, and so on. The very first character has index `-len(s)`. As such, the characters in `"Hello World!"` could also be numbered as follows.

H	e	l	l	o		W	o	r	l	d	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In general, for `0 <= k <= len(s) - 1`, the expression `s[-k]` is shorthand for `s[len(s) - k]`. For example, `s[-1]` selects the `"!"` at the end of the string, and `s[-2]` selects the `"d"` before it.

Concatenation and Repetition

One of the most common string operations is ***concatenation***, which **combines two strings end-to-end with no intervening characters**, and like JavaScript, uses the `+` operator. For instance,



- `"ben" + "jerry"` evaluates to `"benjerry"`
- `"ben" + " and " + "jerry's"` evaluates to `"ben and jerry's"`
- If we've defined variables `SLOne = "eugene"`, and `SLTwo = "sabrina"`, then `SLOne + SLTwo` evaluates to the new string `"eugenesabrina"`.

Unlike JavaScript, however, the **+ operator only works if both operands are strings**, and doesn't convert numbers automat ically to strings. For instance, `"messier" + 87` would crash the Python program, whereas `"messier" + str(87)`, by casting to a string, would evaluate to `"messier87"` (i.e. the name of a  supergiant elliptical galaxy in the Virgo constellation of deep space) without crashing.

Python, intriguingly enough, also **redefines the multiplication operator * for strings to indicate repetition**, so that the expression str * n indicates n copies of the string concatenated together, i.e.

`str * n => str + str + str + ... + str` ***(n times)***

For instance, the expression `"la" * 3` evaluates to `"lalala"`, which is consistent with the mathematical notion that multiplication is just repeated addition. As another cinematic example,

`"la" * 3 + "nd" => ("la" + "la" + "la") + "nd" => "lalaland"`

One niche application is if we want to print a line of 50 hyphens, i.e. `print("-" * 50)`.

String Slicing

Python enables you to **extract a substring** (part of a string) **by specifying a range of index positions** inside square brackets. This operation is known as ***slicing***, which is, I'll say, an oddly surgical term.

There are multiple ways you can specify a slice, including:



- ***str*`[`*start*`:` *stop*`]`. This is the most common one, with ***start*** as the index at which the slice begins, and ***stop*** as **one after** the index at which the slice ends.
- ***str*`[`*start*`:` *stop*`:` *stride*`]`. The third component, called a ***stride***, indicates how many positions are omitted / skipped over between selected characters. If the ***stride*** is negative, the selection goes backwards through the original ***str***'s characters when building out the new substring.

All components of the slice, ***start, stop, stride***, are optional 👀, and have default values when omitted.



- When ***start*** is missing, it defaults to 0, i.e. start at the beginning)
- When ***stop*** is missing, it defaults to `len(str)`, i.e. go to the full length of the string
- When ***stride*** is missing, it defaults to 1, i.e. go through *each* character left to right.

As a pizzeria of tasty string slices, consider the string `ALPHABET = "ABCDEFGHIJK...WXYZ"` ☺.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Let's slice this pizza! Here's the substring result of each of the following slice expressions ☺.

- `ALPHABET[7:9]` `"HI"`
- `ALPHABET[-4:]` `"WXYZ"`
- `ALPHABET[:3]` `"ABC"`
- `ALPHABET[1:-1]` `"BCED...VWXY"`
- `ALPHABET[0:5:2]` `"ACE"`
- `ALPHABET[::-1]` `"ZYX...CBA"`

String Methods

You don't have to commit this entire gallery of string methods to memory, but do be conscious that many powerful tools exist at your disposal for working with Python strings ☺. The most common ones you'll probably utilize are **`find`**, **`rfind`**, **`strip`**, **`lower`**, **`upper`**, **`replace`**, **`isalpha`**, and **`isdigit`**.

Python String Methods : Finding Patterns

***str*`.find(`*pattern*`)`**

Returns the first index of the ***pattern*** substring in ***str***, or returns **`-1`** if the ***pattern*** doesn't appear, e.g.,

- **`"blackpink".find("pink")`** *returns* **`5`**.
- **`"blackpink".find("red")`** *returns* **`-1`**.

b	l	a	c	k	p	i	n	k
0	1	2	3	4	5	6	7	8

***str*`.find(`*pattern, k*`)`**

Same as the one-argument version above, but starts searching from index **`k`** and beyond, which is broadly useful for finding the second or later occurrence of a pattern in a string ☺**.**

***str*`.rfind(`*pattern*`)`**

Returns index of where the ***pattern*** substring **last** occurs in ***str***, or returns **`-1`** if ***pattern*** doesn't appear.

***str*`.rfind(`*pattern, k*`)`**

Same as the one-argument version above, but searches backwards from index ***k*** instead.

***str*`.startsWith(`*prefix*`)`**

Returns `True` if this string starts with prefix, and **`False`** otherwise.

***str*`.endsWith(`*suffix*`)`**

Returns `True` if this string ends with suffix, and **`False`** otherwise.

Python String Methods : Transforming Strings

***str*`.lower()`**

Returns a copy of ***str*** with all letters converted to lowercase.

***str*`.upper()`**

Returns a copy of ***str*** with all letters converted to uppercase.

***str*`.capitalize()`**

Returns a copy of ***str*** with its first character capitalized, and the rest converted to lowercase.

***str*`.strip()`**

Returns a copy of ***str*** with whitespace characters removed from both ends.

***str*`.replace(`*old, new*`)`**

Returns a copy of ***str*** with all instances of the ***old*** substring replaced by the ***new*** substring.

Python String Methods : Classifying Characters
***ch*.`isalpha`()**
Returns `True` if ***ch*** is a letter (e.g., **`a, b,..., z, A, B,..., Z`**, and returns **`False`** otherwise.
***ch*.`isdigit`()**
Returns `True` if ***ch*** is a digit (e.g., **`0, 1, 2, ..., 9`**, and returns **`False`** otherwise.
***ch*.`isalnum`()**
Returns `True` if ***ch*** is a letter or a digit, and returns **`False`** otherwise.
***ch*.`islower`()**
Returns `True` if ***ch*** is a lowercase letter, e.g. **`a, b, ..., z`**, and returns **`False`** otherwise.
***ch*.`isupper`()**
Returns `True` if ***ch*** is an uppercase letter, e.g. **`A, B, ..., Z`**, and returns **`False`** otherwise.
***ch*.`isspace`()**
Returns `True` if ***ch*** is a ***whitespace character***, e.g., a space **`" "`**, a tab **`"\t"`**, or newline **`"\n"`**.
***str*.`isidentifier`()**
Returns `True` if ***str*** is a legal Python identifier (valid name for a variable, function, etc.). For instance, **`"var_one"`** is valid, whereas **`"67"`** (starts with number) and **`"hello world"`** (has space) are not.

Except for **`isidentifier`**, the above methods apply to ***chars***, i.e. strings that happen to be length one.

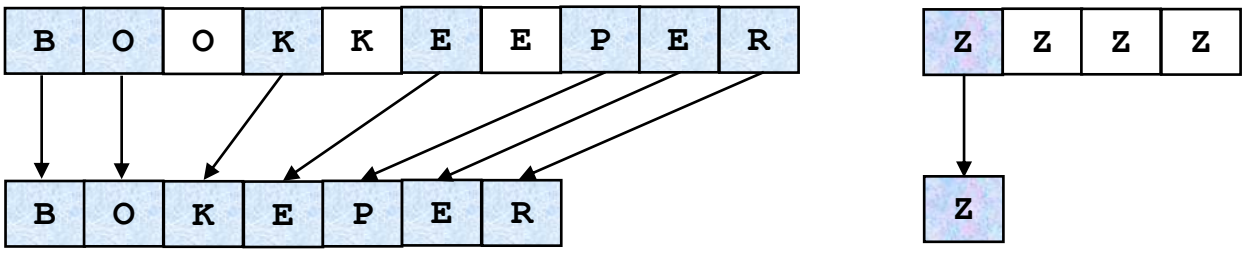
Remove Doubled Letters



Get ready for a history lesson! Are you excited? ☺ Probably not, but here we go! In the early fire-tinged dawn of the 20th century, there was a growing, fearsome blaze of interest in both England and the United States for simplifying—or rather, burning down entirely—the rules used for spelling English words. English is, of course, notorious for its tricky, messy, inconsistent spelling rules, and as the word ***spelling*** itself would imply, some of the rules are just magic, like casting spells and just hoping it works out.

Colonel (“kernel”), *queue* (“q”), *Wednesday* (“wensday”), *island* (“eyeland”), *onomatopoeia* (¯_(ツ)_/¯).

One daring suggestion put forth by the simple-spelling movement was to eliminate all doubled letters, so that ***bookkeeper*** would become ***bokeper***, ***committee*** would contract to ***comite*** (indeed, in life, many committees could be smaller), and the sound of me sleeping, ***zzzz***, would be shortened to just one ***z***.



We'll implement a function, **`removeDoubledLetters`**, which given an input **`word`** string, **returns a new string in which any neighboring identical characters in `s` have been replaced by a single copy**.

When working with string-to-string function tasks, **my over-arching advice is to write out pairs of inputs and outputs**, and look for patterns, e.g., *how letters in the input are mapped to those in the output*, and *how they're transformed if at all*. For the specific diagram on the prior page, you'll notice the following:

- A letter from the input **`str`** is included in the output **only if it doesn't match the letter before it**.
- **If a letter does match its previous letter** (e.g., the second ***o***, ***k***, or ***e*** letter in ***bookkeeper***), **it's a doubled letter**, and exactly what we want to ✂ remove, i.e. should not be in the respelled word.
- **Edge Case:** The first letter (which doesn't have a prior letter) is always included in the new word.

Here's an implementation of **`removeDoubledLetters`** that channels this insight into Python code ☺.

```
def removeDoubledLetters(word):
      result = ""

      for i in range(len(word)):
            ch = word[i]

            # is the first letter in word, or differs from the previous letter
            if (i == 0 or word[i - 1] != word[i]):
                  result += ch      # include it in the respelled word!

      return result
```

Note to prevent **`word[i - 1]`** from crashing with the first letter, we have an **`i == 0`** safety check ☺.

Pig Latin Translation in Python

Before we jump into implementing Pig Latin in Python—as we did in JavaScript weeks ago, it's a canon event at this point for a new programming language—we implement a pair of string helper functions ☺.

```
def isEnglishVowel(ch):
    """ Returns True if ch is an English vowel (A, E, I, O, or U).
    """
    return len(ch) == 1 and "AEIOUaeiou".find(ch) != -1
```

Note that **`find`** returns an index when **`ch`** is found in the vowel string, and returns **`-1`** if it isn't a vowel.

```
def findFirstVowel(word):
    """ Returns the index of the first vowel in the word, or -1 if no vowel can be found.
    """
    for i in range(len(word)):
        if isEnglishVowel(word[i]):
            return i
    return -1
```

As a refresher, here are the rules of ***Pig Latin***, a fun language that hasn't yet been added to the Duolingo app ☹, despite the protests of #1 Duolingo power user and language learning aficionado Jerry Cain ☺.

1. *If the word begins with a vowel*, add **`"way"`** to the end of the word, e.g.,

   orange → ***orangeway,*** ***apple*** → ***appleway,*** and ***elderberry*** → ***elderberryway***

2. *If the word begins with a consonant*, extract the leading consonants up to the first vowel, transport them to the end, and then add the suffix **`"ay"`**.

   scram → ***amscray,*** ***string*** → ***ingstray,*** and ***song*** → ***ongsay***

3. *If the word contains no vowels*, return the word unchanged. To simplify our algorithm, we won't consider ***y*** a vowel here (sorry linguists!). Examples are unnecessary, but for funsies, why not? ☺

   cwm → ***cwm,*** and ***xyzzy*** → ***xyzzy*** (popular magic word in video games)

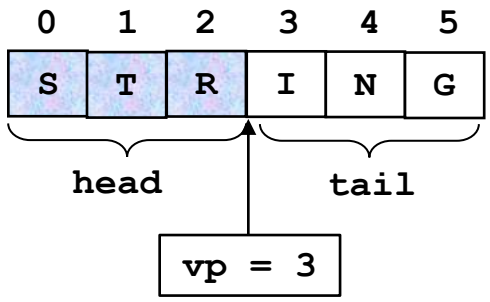
Here's a Python implementation, which uses slicing and concatenation to accomplish the above rules.

```
def wordToPigLatin(word):
    vp = findFirstVowel(word)

    if vp == -1:  # no vowels
        return word

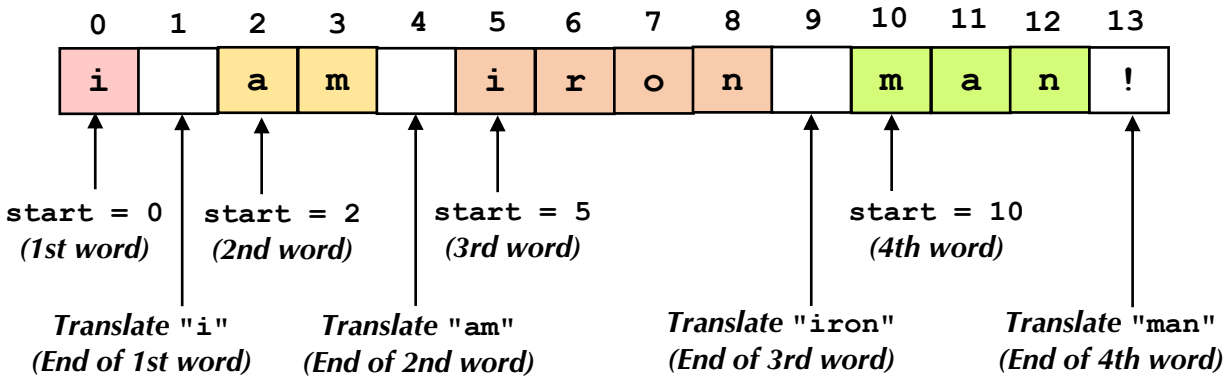
    elif vp == 0:  # word starts with vowel
        return word + "way"

    else:  # word starts with consonant
        head = word[0:vp]    # part of word before first vowel
        tail = word[vp:]     # part of word after and including the first vowel
        return tail + head + "ay"
```



Next, we'll build on the above helper to write a function **`toPigLatin`**, which converts a multi-word string, i.e. a full sentence, from English to Pig Latin. Like the JavaScript version, we do the following:

- Initialize a string **`result = ""`**, which holds the growing, Pig Latin translated string.
- Iterate over **`str`**, use a **`start`** variable to recall the first index of the word we're currently in.
- When we reach a non-letter, the current word ends. We translate it, and add it to the **`result`**, and refresh **`start`** so we can start scanning a new word. **`start = -1`** means no current word.



Using the approach outlined above, here's a more detailed and structured painting of pseudocode.

```
def toPigLatin(str):
    Initialize a variable called result to hold the growing string
    for (each character position in str):
        if (the current character is a letter): # character is a part of a word
            if (we're not yet scanning a word) Remember the start of this word.

        else: # is a separator character, e.g., space, punctuation
            if (we were scanning a word):
                Extract word using the remembered start, and the current end position.
                Call wordToPigLatin to translate the word.
                Append the translated word to the result variable

            Append this separator character to the result variable.

    if (we're still scanning a word at the end):
        Call wordToPigLatin and append the translated word to result.

    Return the result string, and voila, we're done!
```

And lastly, here's the translation of the above pseudocode into a full-fledged Python implementation 🐍.

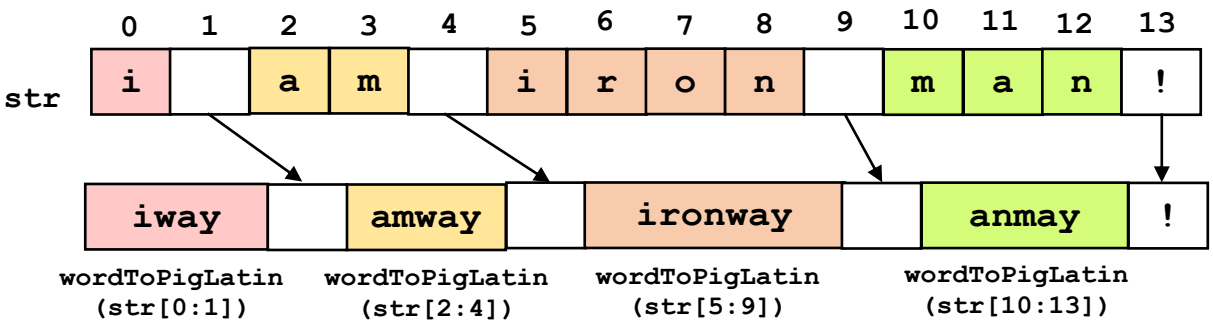
```
def toPigLatin(line):
    """ Converts a multi-word string from English to Pig Latin, [more description here]
    """
    result = ""
    start = -1
    for i in range(len(line)):
        ch = line[i]
        if ch.isalpha(): # letter is part of a word, and if we're not in a word rn, start one!
            if start == -1: start = i

        else: # non-letter
            if start >= 0: # a current one has just ended, so translate it!
                result += wordToPigLatin(line[start:i])
                start = -1
            result += ch

    if start >= 0: # a current one
        result += wordToPigLatin(line[start:])

    return result
```

Let's visually demonstrate how `toPigLatin` would translate the earlier string `"i am iron man!"`.

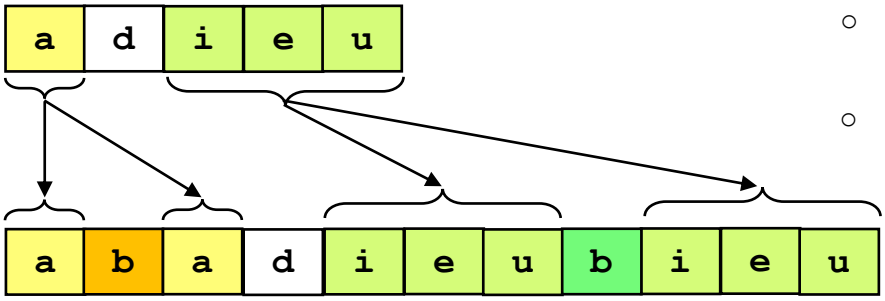


B-Language

Pig Latin is just one of many language games—designed to render spoken words to be incomprehensible to the untrained ear, and to testify to the beautiful, wondrous creativity of our vast world of languages.

Another deeply unserious language game, known primarily in Germany, is ***B-Sprache*** or **B-language**, where words are transformed such that **every vowel cluster is reduplicated with a leading `"b"` letter**.

For example, my favorite goodbye saying, **`"adieu"`**, turns into **`"abadieubieu"`**, and here's why!



- *First vowel cluster is* **`"a"`**, *so becomes* **`"aba"`**
- *Second vowel cluster is* **`"ieu"`**, *so becomes* **`"ieubieu"`**

Our task is to implement the function **`translate(word, sep = "b")`**, which accepts a lowercase English **`word`**, and returns its B-language translation. However, an optional 2nd parameter **`sep`** can be supplied to override the **`"b"`** with any other string. Here are some added examples for your amusement.

- **`translate("quick")`** ⇒ **`"quick"`**
- **`translate("spaghetti")`** ⇒ **`"spabaghebettibi"`**
- **`translate("audiophile")`** ⇒ **`"aubaudiobiophibilebe"`**
- **`translate("queueing")`** ⇒ **`"queueibueueing"`**
- **`translate("jerrycain", sep = "p")`** ⇒ **`"jeperrycaipain"`** ☺ ***hehe sorry Jerry Bear***



Our writeup of the B-language **`translate`** is architected near identically to that of **`toPigLatin`**.

```
def translate(word, sep="b"):
      start = -1  # tracks the beginning index of the current vowel cluster we're in
      translation = ""
      for i in range(len(word)):
            ch = word[i]
            if isEnglishVowel(ch):  # is a vowel, so should be part of a vowel cluster
                  if start == -1:  # no current cluster we're scanning, so start a new one
                        start = i

            else:  # is a non-vowel / separator, so may be end of a current vowel cluster
                  if start >= 0:  # reduplicates this vowel cluster with a leading "b" ☺
                        vowelCluster = word[start:i]
                        translation += vowelCluster + sep + vowelCluster
                        start = -1  # refreshes so we can start a new vowel cluster yay!

                  translation += ch  # adds the non-vowel character as-is

      if start >= 0:  # if we're still in a vowel cluster at the very end, reduplicate it!
            translation += word[start:] + sep + word[start:]

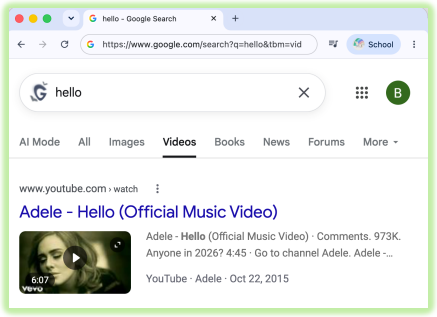
      return translation
```

URL String Extraction

You've almost certainly heard of the acronym URL, but do you know what it actually stands for? ☺

Me neither, until I glanced at today's slide deck. ***Uniform Resource Locators***, more commonly referred to as ***URLs***, are **string expressions used to identify where documents can be found on the vast Internet**.

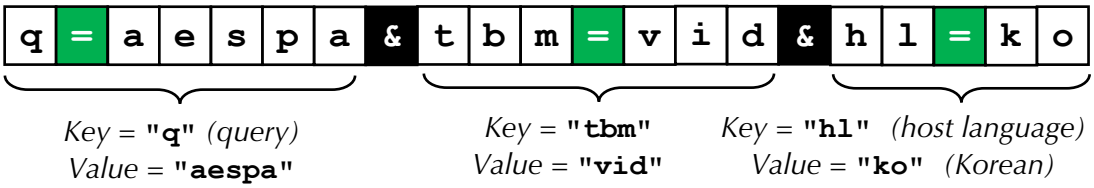
Many URLs include ***query strings***, which are substring fragments appearing after a `"?"` that **further refine the search request**, e.g., what portion or type of document should be retrieved. For example,

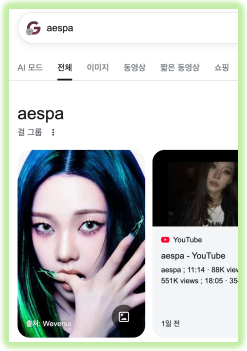


`https://www.google.com/search?q=hello&tbm=vid`

lists all public videos associated with **`"hello"`** (try it out!), which unsurprisingly includes a lot of Adele. The **q** or **query** is **`"hello"`**, and the **tbm** or ***to be matched*** is **`"vid"`**, ostensibly short for video.

Query strings are just stringified versions of **maps**, i.e. a collection of key-value pairs in the sense that you learned about with ***JavaScript aggregates***. In a URL query, keys are separated from values by equal signs (**`=`**), and key-value pairs are separated from one another by ampersands (**`&`**). As an illustration,





All of this is compacted into one string, though according to quantum string theory, everything in the universe is a string ☺. **Our task is to implement the `extract` function, which rips out all key-value pairs embedded in the query string of an input `url`, and prints them out one per line**. For example, the function call

```
extract("https://explorecourses.stanford.edu/search?view=catalog
        &q=CS106X&academicYear=20182019&filter-term-Autumn=on")
```

would publish the following output to the console.

Terminal: Extract

```
Key: "view", Value: "catalog"
Key: "q", Value: "CS106X"
Key: "academicYear", Value: "20182019"
Key: "filter-term-Autumn", Value: "on"
```



Let's implement it! **The first step is to find the question mark `"?"` in the URL**, and take the slice of the string after it. If there's no `"?"`, the URL doesn't have a query, and we can simply return the empty `""`.

You should also know that a query can be bounded by an optional hashtag, after which is something called a fragment identifier, e.g., **`"#page1"`**. This specifies the section / part of the webpage to go to. As such, if there's a `"#"` in the URL, **we take the slice of the URL string between the `"?"` and the `"#"`.**

Here's an implementation of the helper function **getQuery**, which accomplishes exactly the above.

```
def getQuery(url):
    question = url.find("?")
    if question == -1: return ""  # no question mark, so no query substring

    hashtag = query.find("#")
    if hashtag != -1:  # no hash fragment, so take entirety of URL after the question mark
        return url[question + 1:]

    else:  # has hash identifier, so take substring between the question mark and the hashtag
        return url[question + 1:hashtag]
```

For instance, **getQuery("google/search?** q=cats&tbm=isch **#SO CUTE OMGGG")** would return **"q=cats&tbm=isch"** (if you're curious, ***isch*** is short for image search). Who doesn't want cat photos?

Once we've isolated our **query** string, so we should loop through its characters, repeatedly excavating for key-value pairs. We track our current index **start**, with a loop that keeps hamster spinning until start reaches the end, **len(query)**. **In each iteration, we extract and print out the next key-value pair**, by finding the indices of the next **"="** and **"&"** symbols. Here's a helper function that does exactly this.

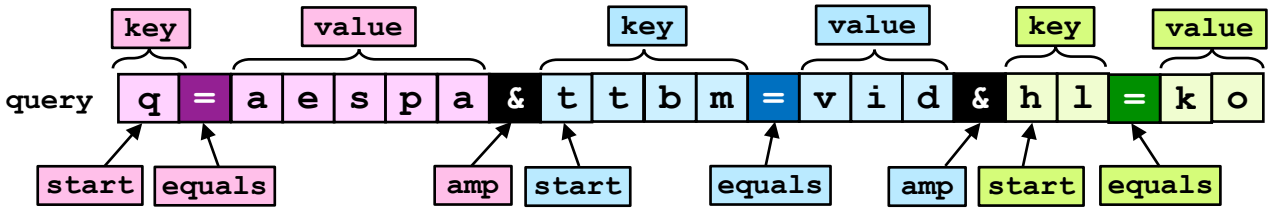
```
def printKeyValuePairs(query):
    start = 0
    while start < len(query):
        equals = query.find("=", start)  # finds equal sign after the start index!
        key = query[start: equals]

        amp = query.find("&", equals + 1)
        if amp == -1:  # no ampersand, so last pair! (value stretches to end of the string)
            value = query[equals + 1:]

        else:  # has another key-value pair after, so value is bounded by ampersand
            value = query[equals + 1: amp]

        print(f"Key: \"{key}\", Value: \"{value}\"")
        start = amp + 1  # next pair (or string end) is located after this "&"
```

This code relies on a ✨ ***Python formatted string*** ✨ for convenience. The string values of **key** and **value** are plugged into **{key}** and **{value}**, respectively, when the **print** is executed. To check your understanding, here's a visual on how this function would crawl through a sample **query** string.



Assembling all the above jigsaw pieces of Python together, we get this insultingly concise writeup ☺.

```
def extract(url):
    query = getQuery(url)
    printKeyValuePairs(query)
```

""" Parses the supplied URL and prints out all the key value pairs in the query, one per line """