Strings

15-110 – Wednesday 09/18

Learning Goals

Use **programming** to specify algorithms to computers

- Index and slice into **strings** as iterable objects to break them up into parts
- Use **built-in methods** to solve problems

String Operations

String Indexing

Last time, we showed how it's possible to find a character in a string based on its location with **indexing**.

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)+5] # error!!
```

Activity: guess the index

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

"b"?

"1"?

"3"?

String Slicing

We can also get a whole substring from a string by specifying a **slice**.

Slices are exactly like ranges- they can have a start point, an end point, and a step. But slices are represented as numbers inside of **square brackets**, separated by **colons**.

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

String Slicing Shorthand

Like with range, we don't always need to specify values for the start, end, and step; there are expected defaults (0 for start, len(s) for end, and 1 for step). But the syntax looks a little different this time.

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

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

`s[:-1]` is the string without the last character

`s[::3]` is the string with every third character

Activity: find the slice

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

String Operations

There are useful string operations outside of indexing and slicing.
We've already seen concatenation:

```
"Hello " + "World" -> "Hello World"
```

We can also use multiplication to repeat a string a number of times!

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

New operation: in

When we have an iterable type (like a string), we can use the **in** operator to check if a value occurs in the string.

```
"a" in "apple" # True!
"4" in "12345" # True!
"z" in "potato" # False!
```

This is much faster to write than our search function from last time!

String Comparison

We can also compare strings, just like how we compare numbers. When we compare two strings, we compare the **ascii values** of each character in order, and the smaller ascii value goes first.

Because the lowercase letters and uppercase letters are listed in order in the ASCII table, we can compare lower-to-lower and upper-to-upper **lexicographically**, the order they'd appear in the dictionary. But that won't work if we compare lowercase to uppercase words!

```
"hello" > "goodbye" # True
"book" < "boot" # True
"APPLE" < "BANANA" # True
"ZEBRA" > "aardvark" # False - lowercase letters are larger!
```

Odds and Ends

We can directly translate from characters to ascii in Python! The function ord(c) returns the ascii number of a character, and chr(x) turns an ascii number into its character.

```
ord("k") # 107
chr(106) # "j"
```

Finally, there are a few characters we need to treat specially in Python. Specifically, these are the enter character (**newline**) and the tab character (**tab**). We can't type these directly into the string, so we'll use shorthand instead:

```
print("ABC\nDEF") # newline, or pressing enter/return
print("ABC\tDEF") # tab
```

The \ character designates that it is a special character, or an **escape sequence**, and it is modified by the letter that follows it. These two symbols are treated as a single character by the interpreter.

String Methods

Using string methods

String methods work differently from built-in functions. Instead of writing:

```
isdigit(s)
```

we have to write:

```
s.isdigit()
```

This tells Python to call the built-in function isdigit() **on the string s**. It will then return a result normally.

Don't Memorize- Use the API!

There is a whole library of built-in string functions that have already been written; you can find them at

https://docs.python.org/3.6/library/stdtypes.html#string-methods

We're about to go over a whole lot of methods, and it will be hard to memorize all of them. Instead, **use the Python documentation** to check the name of a function that you know probably exists.

As long as you can remember which basic actions have already been written, you can always look up the name and parameters at need.

Strings Functions that Return Values

First, let's look at some string functions that return information about the string.

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

`s.count(c)` returns the number of times the character c occurs in s.

`s.find(c)` returns the index of the character c in s, or -1 if it doesn't occur in s.

Using string methods

As an example of how to use string methods, let's write a function that returns whether a string is composed entirely of unique characters.

```
def isUnique(s):
    for c in s:
        if s.count(c) > 1:
            return False
    return True
```

String Methods that Create Strings

Other string methods produce a new string based on the original.

`s.lower()` and `s.upper()` convert a string to all-lowercase or all-uppercase, respectively.

`s.replace(a, b)` produces a new string where all instances of the string a have been replaced with b.

Using String Methods

We can use these new methods to make a silly password-generating function:

```
def makePassword(phrase):
    phrase = phrase.lower()
    phrase = phrase.replace("a", "@")
    phrase = phrase.replace("o", "0")
    return phrase
```

One last method: split()

Finally, we can use another method, `s.split(c)`, to split up a string into a new iterable based on a separator character, c. We'll use this method directly in a for-each loop for now.

```
def findName(sentence, name):
    for word in sentence.split(" "):
        if word == name:
            return True
    return False
```

In-Depth Example: Program an Auto-Grader

Let's solve a slightly more complicated problem. Pretend you're a very lazy professor of underwater basket weaving, and you want to automate the process of grading student essays about the correct process for weaving a basket while underwater.

- What makes a good essay?
- How can we map features of a good essay to features of strings we can calculate?
- How can we check whether our auto-grader is good enough?

Learning Goals

Use **programming** to specify algorithms to computers

- Index and slice into **strings** as iterable objects to break them up into parts
- Use **built-in methods** to solve problems