STRINGS

Objectives:

- How text data is internally represented as a string
- Accessing individual characters by a positive or negative index
- String slices
- Operations on strings: concatenation, comparison, "in"
- Using Python's built-in string functions

Not all data is numerical. The purpose of the string data type is for us to keep track of text information. A string is a collection of any number of characters. And a character is basically anything that you can type on the keyboard, including letters, digits, punctuation symbols, and blank spaces, tabs and newlines. (A newline is a special character that results when you hit the Enter/Return key and signals your computer that it should go to the next line of a text document.)

In Python, we denote a string constant by enclosing it in either single quotes 'like this', or inside double quotes "like this". By convention, most people prefer to use double quotes. You may use either single or double quotes for your string constants, but be consistent.

Python allows variables to hold string values, just like any other type (Boolean, int, float). So, the following assignment statements are valid:

```
name = "Nick Charles"
name = "N"
name = ""
```

In the last example above, there was nothing between the quotation marks, and this is called the empty string, not to be confused with strings containing only invisible characters such as "   " or "   ".

Accessing a character

Since a string is a collection of characters, it makes sense to want to access individual characters or parts of the string. Python uses the brackets [ ] to access portions of a string. Before I show you all the syntax for using the brackets, it is important to understand how the individual characters of a string are numbered.

Inside a string, each of its characters lives inside a cell, and these cells are numbered sequentially from zero. Thus, if s is a string variable and s = "horse", then the individual characters are located as follows:

Index	0	1	2	3	4
Character	'h'	'o'	'r'	's'	'e'

To access an individual character of a string, simply put its index inside brackets after the name of the string. So, s[0] refers to the first character of the string variable s, which in this case is 'h'. Similarly, s[1] is 'o', s[2] is 'r', s[3] is 's' and s[4] is 'e'. Right away, you should notice something about the size of the string and its index values. In this example, our string has 5 characters, and they are located at indices 0..4 (i.e. 0 through 4 inclusive). No matter how long a string is, its first character is always at index 0. The index of the last character is always 1 less than the length of the string.

Think of an index like an apartment number. The letter 'r' is located at index 2.

Interestingly, Python even allows us to use *negative* index values when using the bracket notation. In this case, a negative index number means we are counting from the right end of the string. Thus, the index –1 refers to the last (i.e. rightmost) character of the string. The index –2 is the second from the end, and so on. The nice thing about negative indexing is that you don't need to know the length of the string. The index –1 refers to the last character no longer how big the string is.

The following table illustrates positive and negative indices for the string "doghouse".

Positive index	0	1	2	3	4	5	6	7
Character	'd'	'o'	'g'	'h'	'o'	'u'	's'	'e'
Negative index	–8	–7	–6	–5	–4	–3	–2	–1

This idea of numbering from both ends is common to us in real life. Imagine you are giving directions to find someone's house along a street, or an office along a hallway. For example, let's say that along some hallway there are eight offices. Your office could be, say, the third door on the left, if I come down the hallway from a certain direction. However, if I came from the other direction, you would instead say it's the sixth door on the right. This situation is exactly the same as looking for the letter 'g' in the word "doghouse" above, except that in Python the default numbering starts at zero instead of one.

String slice

Sometimes one character is not enough. A string slice is a collection of several consecutive characters from a string. Again, we use the bracket notation, and specify two values separated by a colon. If s is a string, then the notation s[a:b] means that we want (b – a) characters from the string starting at index a.

For example, if s = "doghouse", then s[2:5] means that we want 5 – 2 = 3 characters starting at index 2 of the string. This means we have "gho". A convenient way to figure out a sting slice is to remember that the notation s[a:b] is analogous to the way we wrote for loops with "for i in range(a, b)". Then, we see that s[2:5] will give us the characters s[2], s[3], and s[4] only. The result of a string slice is another string.

For added flexibility, Python allows us to omit the numbers a and/or b in the notation s[a:b]. Omitting the first number (a) means that we start the string slice at the beginning of the string. Omitting the second number (b) means that we finish the string slice at the end of the string. This is helpful when we don't know exactly how many characters are in the string, or we don't wish to count them.

Examples: if s = "doghouse", then

- s[:3] means we want the first 3 characters of s, which gives us "dog".
- s[3:] means we want everything starting at index 3, which gives us "house".

Incidentally, a string slice can also have an optional third argument representing the stride. In general, the notation s[a:b:c] means to begin the string slice at index a, stop just before index b, and include only the characters at indices that are c apart. In other words, this string slice will include the characters at index a, a+c, a+2c, etc. until we reach index b. For example, if we have s = "doghouse", then s[::3] means to give every third character starting at the beginning, which gives us "dhs".

Continuing our analogy between string slices and the range function, consider this example:

```
for i in range(5, 20, 4):
    print(i)

s = "abcdefghijklmnopqrstuvwxyz"
print(s[5:20:4])
```

Here, the `for` loop will print the numbers 5, 9, 13 and 17. Similarly, the string slice will contain the characters at positions 5, 9, 13 and 17 within our string, resulting in "fjnr".

Concatenation is another fundamental operation on strings. It means to merge two strings into each other. We use the + sign. It works for string variables, string constants, or any combination. Example: "ab" + "cde" results in the string "abcde".

Discussion:

1. In the string s = "abcdefghijklmnopqrstuvwxyz", which letters are located at s[4] and s[–4]?

2. If s = "Salt Lake City", then what string slice equals just the word "Lake"?

3. What is the output of the following code?

```
s = "dolphin"
print (s[1] + s[4])
print (s[4] + s[1])
print (s[1:4])
```

4. Let s = "cat" and t = "house". Create an expression using s and t that equals "house cat".

Changing a string

There is a technical point about strings in Python. String objects cannot be changed. If you have a string variable, and you just want to change one of its characters, you cannot simply write an assignment statement to just a character. You have to assign to the entire string.

Going back to s = "doghouse", if we subsequently decided to start s with a capital letter D, the following statement is illegal: s[0] = 'D'. To change one letter of a string, we can do the following:

- Create a slice of the string that excludes the first character.
- Create a new string starting with the letter "D", and concatenate with the old string slice.

The resulting assignment statement becomes: s = "D" + s[1:]

Here is another example. Suppose s is a string containing several words, and we want to modify s by appending a period at the end of the string. Here is how to do it:

```
s = s + "."
```

Incidentally, the above assignment statement can also be rewritten using a shortcut assignment operator +=. Just as + has different but analogous meanings for numbers and strings, so too for +=. If x and y are strings, then the statement x += y means to change x by concatenating y onto the end of x. The string y is unchanged. For example, this code:

```
word = "dog"
word += "s"
```

results in the word "dogs" being stored in the variable called word.

Important operations on strings

You have just seen concatenation of string, but there is more.

We can compare two strings using relational operators, and thereby compare them just like we compare numbers. Python relies on Unicode values of each character to determine alphabetical order. For instance, 'a' is considered the "lowest" letter of the alphabet, and 'z' is the "highest" letter. So, 'a' < 'b', ‘b’ < ‘c’, and so on. Another way to think of < is like this: if s1 and s2 are strings, then s1 < s2 means that s1 should appear earlier in the dictionary than s2. For example, “cat” is less than “dog”.

Long ago, someone made the arbitrary decision that capital letters are “less than” lowercase letters. Consequently, “Cat” < “cat”. Also, the absence of a character is less than any real character. For example, “dog” is less than any 4-character string that begins with “dog”. In particular, “dog” < “dogs”.

Another useful operator is the Python keyword `in`. It is used to see if a letter or substring exists in a string. It returns True or False. For example, “x” in “aeiou” returns False.

Important string functions

Functions are written by people to save us time and effort in coding. Our Python installation has several useful built-in functions for strings. Here are some of the most commonly used string functions.

Function	Meaning	Example
len(s)	How many characters in s	len(“dog”) returns 3
s.find(<letter or pattern>)	Returns index of first occurrence of the letter or pattern	s = “dog” s.find(‘o’) returns 1
s.upper( )	Returns s with all of its letters converted to uppercase. Does not change s.	s = “Dog” s.upper( ) returns the string “DOG”
s.lower( )	Analogously returns s with all of its letters converted to uppercase. Does not change s.	s = “Dog” s.lower( ) returns the string “dog”
s.replace(<old char>, <new char>)	Returns a new string where every occurrence of the old char is replaced with the new char. Does not change s.	s = “strings” s.replace(“s”, “oa”) returns the string “oatringoa”
s.count(<letter or pattern>)	Returns how many times the substring appears in s.	s = “the cat in the hat” s.count(“at”) returns 2

There is no function to reverse a string, but we can accomplish this if we use –1 as the third parameter of a string slice. In other words, s[:: –1] is the reversal of the string s.

Exercise: Suppose we just got a word from the user, and it is now stored inside a string variable s. How would we determine the following?

1. If it contains the letter T, capital or lowercase
2. The number of times the capital letter T appears
3. The location of the first capital T
4. The location of the second capital T, assuming that it exists
5. If it contains a digit
6. If it contains at least 2 vowels

7. The number of vowels
8. The locations of all the vowels
9. If it has more than 5 characters
10. The fifth character
11. The last 3 characters
12. If the first and last characters are the same
13. If all of its letters are capitalized

More questions about strings: You should be able to work each problem out by hand without using the computer, but feel free to check your answer with the Python system.

14. Suppose `s.find("g")` returns 3. What can we conclude about the string s?

15. Suppose `s.count("g")` returns 3. What can we conclude about the string s?

16. What is the output of the following code?
```
word1 = "well"
word2 = "done"
word1 += word2
print (word1)
```

17. What is the string contained in s2 when the following code finishes?
```
s1 = "dolphin"
s2 = ""
for c in s1:
    if c in "aeiou":
        s2 += c
```

18. What is the output of the following code?
```
s1 = "dolphin"
s2 = ""
s3 = ""
for c in s1:
    if c in "aeiou":
        s2 += c
    else:
        s3 += c
print (s2, s3)
```

Cryptography

As you know, a string is a series of characters. It turns out that every character that can be typed or printed out has an internal numerical representation. This representation is called ASCII code (or Unicode). For example, the capital letter 'A' is represented inside the computer by the number 65. Since character data is essentially numerical data, we can perform mathematical functions on our characters. This capability is exploited most famously in the world of cryptography.

In Python, there are two useful functions that we would use in order to support cryptography:

`ord`(character) → ASCII code

`chr`(ASCII code) → character

For example, ord('B') returns 66, and chr(66) returns 'B'.

Here is a simple example illustrating how cryptography works. Let's suppose we wish to encrypt a string by adding the number 3 to all of its characters. The pseudocode would work as follows:

```
for each letter in the string:
    value = ord(letter)
    print out the encrypted letter chr(value + 3)
```

Practice

Hint: I think you will find that several problems will have this basic structure: a loop containing some kind of `if` statement, so that you can "interrogate" each letter of a string.

1. Ask the user for a string, and print out every fiftieth character of the string. In other words, the characters at index 0, 50, 100, 150, etc.

2. Ask the user for a word. Determine if the word contains a double letter. A double letter means that you have 2 consecutive characters that are the same. Hint: a word is simply a list or sequence of characters. If you feel more comfortable with numbers, this problem is the same as checking to see if 2 consecutive numbers in a list are the same.

3. Suppose s is a string variable containing a time, such as "6:17". There is an integer on either side of a colon. Assume this represents minutes and seconds. Output the total number of seconds this represents. For example, in this case 6:17 equals 377 seconds.

4. Suppose we have two strings that both represent an amount of time given in minutes and seconds. You may assume there is an integer on either side of a colon. Find the sum of the two times, and print this in minutes and seconds as well. Be sure to "carry" if the sum of the seconds is 60 or more.

5. Ask the user to enter the date, in the form of month/day. Determine if this is a valid date. For example, 1/45 is invalid. You may assume it is not a leap year.

6. A computer system has a rule about user names. They must be 8 characters long, and must contain at least 2 letters and at least 2 digits. Given a proposed user name, test if it's valid.

7. Ask the user to enter a string. Determine if this input is a valid identifier in Python. In other words, the first character is a letter or underscore, and each other character is a letter, underscore or a digit.

8. Suppose s is a string variable. How would you capitalize all of the vowels in this string?

9. Suppose s is a string containing a first and last name. Print out this name, with the last name followed by the first name. For example, if s == "Al Capone" you should print "Capone Al".

10. Knowing that the ASCII code of 'a' is 97 and the ASCII code of 'z' is 122, how would you write a loop to print the alphabet like this: abcdefghijklmnopqrstuvwxyz?

11. Suppose s is a string containing a phrase. Assume that the phrase has no punctuation. Traverse this string, and print out the length of all of its words. For example, if s equals "Back to the future", then we should output the numbers 4 2 3 6.

12. Design a game where the user has to enter words repeatedly, and after the first word, each word must begin with the same letter that its previous word ended with. The game ends when the player fails to do this correctly. The game should count how many proper words were entered.

13. Let's consider Roman numerals.
    a. Show how we can design a program that asks the user to enter a positive integer less than 4000. The program will then output this number using Roman numerals.
    b. Show how we can design a program that converts a string of Roman numerals to an integer.