CptS 111, Spring 2023
Lect. #4, Jan. 23, 2023
Class Notes

Today's Agenda:

1. More on strings
2. ASCII characters
3. `chr()` and `ord()` functions

Postscript: Using Dot Notation with Modules

I forgot to emphasize last week that when we import modules the way we did, we must use the name or alias of the module together with dot notation. For example, we used:

```
import math
math.pi
math.factorial(5)
math.sqrt(25)
```

Ch. 3

1. String Basics

Recall that strings are just sequences of alphanumeric characters. We've used strings as literals, e.g., `pet = 'cat'`, but now we're going to learn a little more about them. In Ch. 8, we'll learn even more.

A. String Indexing: Because strings are sequences, we can use indexing to identify characters in them; each character is an element in a string, and indexing begins with `0`, that is, the first character in a string has an index of `0`. We always enclose an index in square brackets.

In [1]:
```
# In Python, indexing _always_ begins with 0

name = 'strong bad'
print(name[3])
```

```
o
```

We can use negative indexing, i.e., negative integers. Indexing begins with `-1` and starts at the end of the string. Thus, `[-1]` gives the last character in a string. This may seem a little strange, but negative indexing can be very useful, especially when we don't know the length of a string. If you struggle with this concept, note that we index the first character of a string with `0` and continue adding `1` as we move to the right. To determine negative indexing, again start the first character with an index of `0` but then move left. The number to the left of `0` is `-1`, but there isn't a character there, so instead wrap around to the end of the string, and index the last character in the string with `-1`. Then the next to the last character has an index of `-2`, and so on.

In [2]:
```
# Negative indexing can be very useful as you'll see later

print(name[-1])  # Indexing the last chr as -1 is especially useful!
```

```
d
```

B. Immutability of Strings: Strings are **immutable**, which means that we can't change them, e.g.,

In [3]:
```
# Strings can't be changed

cat = 'mouse'
cat[0] = 'h'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [3], in <cell line: 4>()
      1 # Strings can't be changed
      3 cat = 'mouse'
----> 4 cat[0] = 'h'

TypeError: 'str' object does not support item assignment
```

However, we can assign a different string to a given lvalue:

In [4]:
```
# We can assign a different value to an lvalue we
# used before

cat = 'dog'
print(cat)
```

```
dog
```

C. `len()` Function: We use the `len()` function to find the length of a string, i.e., how many characters are in the string. Note that spaces are characters and so are punctuation marks, tabs, and newlines!

In [5]:
```
# Use len() to determine the number of characters in
# a string, including blank spaces and punctuation
# marks which are all characters (notice the use of
# nested functions!)

print(len('How long am I?'))
```

```
14
```

D. Operator Overloading: In Python, math operators can be used in non-math operations. We can use this "operator overloading," both repetition (*) and concatentation (+) with strings.

In [6]:
```
# "Multiplication" (*) will produce n number of
# characters

stars = '*' * 50   # Repetition
print(stars)
print(len(stars))
```

```
**************************************************
50
```

Note that concatenation (+) will place two objects immediately next to each other.

In [7]:
```
# Recall that input() returns a string; a single
# character is a string!

num1 = input('Enter an integer: ')
num2 = input('Enter an integer: ')
print(num1 + num2)  # Concatenation
```

```
Enter an integer: 4
Enter an integer: 2
42
```

In [8]:
```
# If we want spaces between words, we have to add them

sentence = 'This' + ' ' + 'is' + ' ' + 'a' + ' ' + 'sentence.'
print(sentence)

# If we don't, everything will be smooshed together
print('This' + 'is' + 'a' + 'smooshed' + 'sentence.')
```

```
This is a sentence.
Thisisasmooshedsentence.
```

E. Use of Quotes: As we stated last week, we create string literals using single or double quotes, e.g.,

In [9]:
```
# Note the use of quotes and the escape character (\)

print("We can use 'single quotes' inside double quotes.")
print('We can use "double quotes" inside single quotes.')
```

```
We can use 'single quotes' inside double quotes.
We can use "double quotes" inside single quotes.
```

In [10]:
```
# We can also use escape characters for quotes inside quotes.

print('\'Single quotes\' inside single quotes with the escape character \\.')
print("\"Double quotes\" inside double quotes with the escape character \\.")
```

```
'Single quotes' inside single quotes with the escape character \.
"Double quotes" inside double quotes with the escape character \.
```

F. Initializing an Empty String: We initialize an empty string using a pair of single or double quotes. We have to do this when we want to add a character to a string (and, as we'll see later, there are many times this is useful!). **If I instruct you to initialize an empty string, this is what I want you to do.**

In [11]:
```
# Empty strings can be very useful; they're very easy to
# initialize

i_am_empty = ''
print(i_am_empty)
```

To add a character to an empty string, we use augmented assignment and operator overloading. Yep! We can combine the two because Python rocks! Note that strings are still immutable, but by using augmented assignment we don't change the string; we reassign it!

In [12]:
```
# An example of why Python is so cool, i.e., we can
# combine augmented assignment and operator overloading.

# Augmented assignment combined with operator overloading,
# i.e., using + to concatenate.
acronym = ''   # Initialize empty string
acronym += 'T' # Shorthand for acronym = acronym + 'T'
acronym += 'M'
acronym += 'I'
print(acronym)
```

```
TMI
```

2. ASCII Characters

Recall that we communicate with computers using 1's and 0's. A single bit (recall bit stands for `b` inary dig `it`) can represent a 1 (on) or a 0 (off). Historically, everything has been done using bytes where one byte = 8 bits. ASCII characters use only 7 bits. How many different characters can be represented by 7 bits? `2 ** 7 = 128` (0-127); therefore, there are 128 ASCII characters.

```
0-31:    unprintable characters (can be whitespace, e.g., a tab)
32-127:  printable characters (32 is a blank space in ASCII)

65-90:   uppercase alphabet, A-Z
97-122:  lowercase alphabet, a-z (uppercase < lowercase)
```

Note that 0-127 are the decimal representations of the actual binary representations of characters. For example, the binary number 1000001 is 65 in decimal and represents `A` , and the binary number 1100001 is 97 in decimal and represents `a` . In addition, recall that Python uses a backslash (`\`) to indicate an escape sequence. Each escape sequence represents a particular character command:

```
\\   print backslash
\'   print single quote
\"   print double quote
\t   tab
\n   newline
\f   form feed
\r   carriage return
\v   vertical tab
\b   backspace
\a   alert
```

Some of these aren't really used any longer because they represent old technology. The first five are still common.

As mentioned earlier, blank spaces in a string are indexed and contribute to the length of a string.

In [13]:
```
# A blank space is a character as are \n, \t, and so on

nums = '0 1234'
print('3\n\t2\n\t\t1')
print('len(nums):', len(nums))
```

```
3
	2
		1
len(nums): 6
```

3. `chr()` and `ord()` Functions

- `chr()` gives ASCII character for integer argument (decimal default)
- `ord()` gives ASCII value (decimal) for character input

```
In [14]: # chr() takes an integer argument and returns a
         # character

         print(chr(42))
```

```
*
```

```
In [15]: # ord() takes a character argument and returns
         # an integer

         print(ord('A')) # Note the use of quotes
```

```
65
```

`chr()` and `ord()` are inverse functions of each other.

```
In [16]: # ord() and chr() are nested functions; first chr() returns a character;
         # then ord() returns an integer

         print(ord(chr(42)))
```

```
42
```

```
In [17]: # chr() and ord() are nested functions

         print(chr(ord('*'))) # Inner function executed first
```

```
*
```

Next, let's check out the ASCII printable characters:

```
In [18]: # We'll learn for-loops in Ch. 6; no need to worry about them now
         # Note that we initialize an empty string and then we used augmented
         # assignment w/ operator overloading in the loop

         s = ''                     # Initialize empty string
         for num in range(32, 128): # loop through num values
             s += chr(num)          # add chr(num) to s
         print(s)                   # s is a sequence of chr's, i.e., a string
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

The first character is a blank space, i.e., `32` is the ASCII number for a blank space. Now let's get back the ASCII numbers for the ASCII printable characters:

```
In [19]: # We'll learn two different varieties of for-loops;
         # here we use the second variety

         for ch in s:
             print(ord(ch), end=' ')
```

```
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 10
7 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
```

We can also use `chr()` and `ord()` with the extended Unicode which uses up to 32 bits for characters, i.e., `2 ** 32 > 4` billion characters. That's a lot of characters! These actually include emojis, and there's a committee that decides on characters to add, including emojis.

```
In [20]: # Now we're using Unicode which allows many more
         # characters (> 4 billion!)

         t = ''
         for num in range(14000, 14301):
             t += chr(num)
         print(t)
```

```
㚰㚱㚲㚳㚴㚵㚶㚷㚸㚹㚺㚻㚼㚽㚾㚿㛀㛁㛂㛃㛄㛅㛆㛇㛈㛉㛊㛋㛌㛍㛎㛏㛐㛑㛒㛓㛔㛕㛖㛗㛘㛙㛚㛛㛜㛝㛞㛟㛠㛡㛢㛣㛤㛥㛦㛧㛨㛩㛪㛫㛬㛭㛮㛯㛰㛱㛲㛳㛴㛵㛶㛷㛸㛹㛺㛻㛼㛽㛾㛿㜀㜁㜂㜃㜄㜅㜆㜇㜈㜉㜊㜋㜌㜍㜎㜏㜐㜑㜒㜓㜔㜕㜖㜗㜘㜙㜚㜛㜜㜝㜞㜟㜠㜡㜢㜣㜤㜥㜦㜧㜨㜩㜪㜫㜬㜭㜮㜯㜰㜱㜲㜳㜴㜵㜶㜷㜸㜹㜺㜻㜼㜽㜾㜿㝀㝁㝂㝃㝄㝅㝆㝇㝈㝉㝊㝋㝌㝍㝎㝏㝐㝑㝒㝓㝔㝕㝖㝗㝘㝙㝚㝛㝜㝝㝞㝟㝠㝡㝢㝣㝤㝥㝦㝧㝨㝩㝪㝫㝬㝭㝮㝯㝰㝱㝲㝳㝴㝵㝶㝷㝸㝹㝺㝻㝼㝽㝾㝿㞀㞁㞂㞃㞄㞅㞆㞇㞈㞉㞊㞋㞌㞍㞎㞏㞐㞑㞒㞓㞔㞕㞖㞗㞘㞙㞚㞛㞜㞝㞞㞟㞠㞡㞢㞣㞤㞥㞦㞧㞨㞩㞪㞫㞬㞭㞮㞯㞰㞱㞲㞳㞴㞵㞶㞷㞸㞹㞺㞻㞼㞽㞾㞿㟀㟁㟂㟃㟄㟅㟆㟇㟈㟉㟊㟋㟌㟍㟎㟏㟐㟑㟒㟓㟔㟕㟖㟗㟘㟙㟚㟛㟜
```

As we can see, the Unicode numbers 14000-14300 represent Chinese characters.

In [21]:
```
# We'll end our lecture with another Unicode character.

p = chr(128169)
print(p)
```