In Python, strings are a sequence of characters surrounded by single or double quotes. As such, numbers surrounded by quotes are strings. However, if you want to have a string that represents a quote of a person speaking (such as 'Joe says “Hello!” to Fred.'), then single quotes are required on the outside.

As with some other languages, the + operator can be used to concatenate strings. So:

print (“abc” + “def”) → abcdef

The repetition operator is the * used with strings, which has a multiplying affect:

print (3 * “z”) → zzz

Of course, strings can be assigned to variables.

str = “abcdef”

We can refer to the character of a string via its index.

print str[2] → c

And in reverse (from the end):

print str[-2] → e

The length of a string:

print len(str) → 6

For substrings:

print str[3:5] → de

There are the Boolean 'in' and 'not in' operators for comparing strings.

print ('abc' in str) → True
print ('xyz' not in str) → True

String and character methods
These include methods for counting a character's occurrences in a string, changing case, returning index of a char, replacing characters with a substitute, etc.

Python uses ASCII encoding. ASCII characters correspond with decimal numbers, as can be referenced from ascii_table.txt (see in our class D2L site: Materials / Content / Reference documents).
(While on the topic of ASCII, checkout https://en.wikipedia.org/wiki/ASCII_art for ideas on what can be generated with characters and a programming language.)

Using Python's character functions, ASCII characters can be interpreted in different ways:

```
print ( ord('A') )        → 65
print ( char(65) )        → A
print ( '12345' == str(12345) )        →        True
```

Tools like these make it easy to encode and decode messages, a first step towards encryption and decryption. We can create such algorithms of our own, or rely on Python libraries that exist for such, including AES, Crypto and hashlib. There are a variety of interesting ciphers that you can use to encrypt a message.

Data type conversion
When a program reads data from external resources, such as a text file, the initial data type is typically a string, while the format might be SQL, JSON, XML, or CSV. Thus, data must be converted to a type that is appropriate for desired operations. Python has data type conversion functions to assist with this purpose. Here is a sample of such functions:

```
chr(n)     →  number to character
float(o)     →              object to floating-point
hex(int)      →              integer to hexadecimal string
int(o [,base])   →              object integer
long(o [,base] )   →              object to long integer
oct(int)      →              integer to octal string
ord(c)      →              single char to integer
str(n)      →              number to string
unichr(int)     →              integer to Unicode char
```

Python also has an interesting string parsing function called eval that works like an in-line interpreter. It can evaluate a mathematic expression in a string, such as:

```
a = 1
eval('2 + a')   →              3
```

Similarly, the repr function converts an object to an expression string.

f-string literals
These are a recent Python feature, since version 3.6. This works such as:

```
num1 = 4
num2 = 5
print(f"The sum of {num1} and {num2} is {num1+num2}.")
```
→ The sum of 4 and 5 is 9.

Note that evaluation takes place within the {} curly brackets.

Traditional C-like string formatting

The string format operator % is used like that in the printf functions of C and Java.

```
day=1
month='June'
year=2020
print ('The date is %s %i, %i' % (month, day, year))
```
→ The date is June 1, 2020

Selected format symbols for %:

Symbol	Application
%c	character
%s	string
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lower case)
%e	exponential notation (lower case e)
%f	floating point real number

Selected built-in string functions

(more here: https://www.w3schools.com/python/python_ref_string.asp)

capitalize( ) – capitalize first letter of string

center(n, c) – create string that is padded with n chars c on each side of the string.

count(str, i, n) – occurrences of substring str from index i to index n.

find(str, i, n) – if substring str occurs in index range i to n, return the first index, or -1 otherwise.

isalpha() – return true if string has at least 1 character and all characters are alphabetic; false otherwise.

isdigit() – Return true if string contains only digits; false otherwise.

len(str) – return the length of the string str.

lower() – converts all uppercase letters in string to lowercase.

lstrip() – remove all leading whitespace in string.

replace(old, new [, max]) – replace all occurrences “old” with “new” (for max occurrences if given).

split(del) – splits a string along the delimiter del, returning a list of substrings

strip(str) – performs both lstrip() and rstrip() on string.

Regular expression pattern matching
Regular expressions are somewhat standard across programming languages.

We use Regular Expressions for text processing in terms of:

- •
- validation: check if a string matches a pattern; have a boolean (true/false) result
- •
- filtering: find substrings in a string that match a pattern
- •
- substitution: replace substrings in a string that match a pattern
- •
- split: collect in a list substrings that match a pattern

Some pattern matching metacharacters:
. matches a single character
/ search/find
^ matches the beginning of the line
$ matches the end of the line
\ removes special meaning for the character
\d match any digit
[ ] matches enclosed characters
[^] matches characters that are not enclosed
* zero or more occurrences of previous regular expression
( ) Create a substring to match; create groups
{ } Match a specific number of times {at least, at most} {exactly} {at least,} {,at most}
? Match zero or one time max
+ Match at least once
| alternation – OR

Some examples:
. xyz. any character, X, Y, Z, any character
/ the/ find a space followed by “the”
^Leo find the word Leo if it is at the beginning of a line
$Leo find the word Leo if it is at the end of a line
\. makes . become a period, not a match character
[Tt]he matches both the and The
[A-Za-z] matches all uppercase and lowercase alphabetic characters
[^0-9] matches any non-numeric characters
A* zero or more consecutive A's
AA* one or more consecutive A's
.* zero of more characters

A few POSIX character classes:
[:alnum:] The alphanumeric characters; in ASCII, equivalent to [A-Za-z0-9]
[:alpha:] The alphabetic characters; in ASCII, equivalent to [A-Za-z]
[:lower:] The lowercase letters

ASCII is a subset of Unicode. So, ASCII characters and Unicode characters have the same numbering. Most Unicode characters are treated as literals, such that "A" means "A". However, for metacharacters to be treated as literals, they need to be proceeded by a backslash \ or surrounded by half-quotes (this is termed escaping). For example, with \([A-Za-z]+\) the left and right parenthesis characters become part of the pattern sought instead of being metacharacters used for grouping. Similarly, to get a literal backslash, we have to put another backslash before it, such as \\.

Python uses regular expressions as components of certain functions. Use the re library via "import re" to get these. (See more here: https://www.w3schools.com/python/python_regex.asp)

For example:

re.match(pattern, string, flags=0) – match pattern in string; return only matched data (flags optional)
re.search(pattern, string, flags=0) – search for first occurrence of pattern in string.
re.sub(pattern, repl, string, max=0) – replace all occurrences of pattern in string with repl for max
occurrences (if provided).

Formatting and parsing examples
Knowing how to parse and format a block of text is useful for understanding how to process an external data file.

First of all, we can create a data table for display in the terminal. That table could later be stored in a file or even a variable.

```
#ascii-table.py

# print header
print("Decimal\tASCII\tBinary\tOctal\tHexadecimal")

# get the capital letters of ASCII tables
for i in range(97, 122):
  ch = chr(i)    # base10 integer to ASCII char
  bn = bin(i)    # base10 -> base2
```

```
  oc = oct(i)     # base10 -> base8
  hx = hex(i)     # base10 -> base16

  # concatenate the various base representations into one string
  # then convert that string to up case (ignoring the literal table \t
  outstr = f"{i}\t{ch}\t{bn}\t{oc}\t{hx}".upper()

  # display
  print(outstr)
```

Run it:

```
python3 ascii-table.py  →

Decimal  ASCII Binary   Octal  Hexadecimal
 97    a  0b1100001  0o141 0x61
 98    b  0b1100010  0o142 0x62
 99    c  0b1100011  0o143 0x63
 100   d  0b1100100  0o144 0x64
 101   e  0b1100101  0o145 0x65
 102   f  0b1100110  0o146 0x66
 …
 …
```

Here is another example:

```
# cosmicrays.py
# a crude example of some explicit operations
import re

# here is our hard-coded data (source: http://cosmicrays.oulu.fi )
# note the multi-line string between """ and """ ... the first line is the header
data =
"""Timestamp,FractionalDate,UncorrectedCountRate[cts/min],CorrectedCountRate[cts/min],Pres
sure[mbar]
2020-05-22T00:00:00Z,143.0000000,5723,6778,1022.63
2020-05-23T00:00:00Z,144.0000000,5972,6789,1017.09
…
…
2020-06-12T00:00:00Z,164.0000000,5660,6810,1024.75
2020-06-13T00:00:00Z,165.0000000,5646,6761,1024.12
2020-06-14T00:00:00Z,166.0000000,5845,6736,1018.93
2020-06-15T00:00:00Z,167.0000000,6060,6736,1014.03
2020-06-16T00:00:00Z,168.0000000,5922,6752,1017.49
```

```
2020-06-17T00:00:00Z,169.0000000,6009,6764,1015.75
2020-06-18T00:00:00Z,170.0000000,6282,6764,1009.74
2020-06-19T00:00:00Z,171.0000000,5967,6746,1016.37
2020-06-20T00:00:00Z,172.0000000,5734,6745,1021.69
2020-06-21T00:00:00Z,173.0000000,5816,6761,1020.12
2020-06-22T00:00:00Z,174.0000000,6085,6765,1014.06"""

# to get usable data out of such a string we must heed two delimiters:
# a comma ',' and a hidden new-line character '\n'
setOfLines = data.split("\n")

# this printing still does not parse by line... see, it needs another split
print(setOfLines)

# now printing looks much better in the following loop
for line in setOfLines:
  print(line)

# now parse out data with the ',' delimiter
# lets get the pressure reading on June 12th
lineCounter = 0;

for line in setOfLines:
  if(lineCounter != 0):
    lineData = line.split(",")     # split the line into an array of strings
    tmp1 = lineData[4].replace("\n", "")   # last element has EOL char attached; replace it with
blank
    tmp2 = tmp1.strip()               # remove R and L empty strings if they exist
    tmp3 = float(tmp2)           # convert to float for calculations and comparisons

    # if the data is June 12th, print element 4
    if (re.search("6-12", lineData[0])):
      print("Pressure reading on June 12th: ", tmp3)
  lineCounter = lineCounter + 1    # increment the line counter
```

Run it:

```
python3 cosmicrays.py
→
…
…
2020-06-21T00:00:00Z,173.0000000,5816,6761,1020.12
2020-06-22T00:00:00Z,174.0000000,6085,6765,1014.06
Pressure reading on June 12th:  1024.75
```

Data conditioning
Before information can be extracted from data, the data itself must be in some standard processable form. For example, problems may include:

- •
- missing values
- •
- incorrect format
- •
- erroneous entries
- •
- hidden characters
- •
- inconsistent field delimiters
- •
- values from other records

Data pre-processing consists of using regular expressions for text substitution, deleting faulty records, auditing random data, and interpolating data from valid records to substitute for data in erroneous records.

Where the types of missing data are understood in advance, you can devise a program to handle such data conditioning. While you might be tempted to correct the data by hand, what if the data set is huge, say the equivalent of a textbook, or a library of textbooks? Furthermore, the data source may be continuous or streaming, so data conditioning has to be processed in real-time using temporary storage of data segments.

How to deal with missing data:
Records that are blank or contain other than acceptable entries are indicative of missing data. One solution is to interpolate the data of adjacent records. For example, given consecutive missing days of data in a time series, from day N to day M, create a median value at day N+(M-N)/2. Use the average of the two days N-1 and M+1 and assign this value to said median. So, (value[N-1] + value[M+1]) / 2 goes at place N+(M-N)/2. Repeat the procedure until all gaps in the data are eliminated.

Standardizing data:
By the time we get to the topic of collections, we will see standardization methods which depend on the measures of dispersion of a collection. A common technique is to get the maximum and minimum values from a series, and then consider a selected reading in relative terms, such as:

relative_value = (selected_reading – min) / (max - min)