# ECS 32A - Strings

*Aaron Kaloti*

UC Davis - Summer Session #1 2020

# String Arithmetic

- Strings support `+` and `*`.

```
>>> print("abc" + "def")
abcdef
>>> print("abc" + "def" + "ghi")
abcdefghi
>>> print("abc" * 2)
abcabc
>>> print("123" * 3)
123123123
>>> print("abc" + " " * 2 + "def")
abc  def
```

- We refer to use of the `+` with strings as **string concatenation**.

# str()

- `str()` converts its input to a string.

```
>>> str(1)
'1'
>>> str(1.8)
'1.8'
>>> type(3)
<class 'int'>
>>> type(str(3))
<class 'str'>
```

# `len()`

- `len()` is a built-in function that allows the caller to find the length of a string.

```
>>> print(len("abc"))
3
>>> print(len("abcde"))
5
>>> print(len(""))
0
```

# Indexing

- Use **indexing** to access an individual **character** within a string.

```
>>> name = "Aaron"
>>> print(name[0])
A
>>> print(name[1])
a
>>> print(name[2])
r
>>> print(name[3])
o
>>> print(name[4])
n
```

- It is possible to give an index that is *out of bounds*:

```
>>> print(name[5])
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    print(name[5])
IndexError: string index out of range
>>> print(name[6])
    ...
IndexError: string index out of range
```

# Zero-Based Indexing

- Notice that indexing starts at 0, *not* 1. This is known as **zero-based indexing**, as opposed to one-based indexing.
  - Many languages use zero-based indexing, including: C++, C, Java, and JavaScript.
  - Some languages instead use one-based indexing, including R and MATLAB.
    - As you can imagine, this gets annoying once you have learned multiple programming languages.

# Indices Must be Integers

```
>>> print(name[2.3])
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    print(name[2.3])
TypeError: string indices must be integers
```

# Negative Indices

- Python also supports negative indices, for "counting backwards" from the end of the string.

```
>>> print(name[-1])
n
>>> print(name[-2])
o
>>> print(name[-3])
r
>>> print(name[-4])
a
>>> print(name[-5])
A
>>> print(name[-6])
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    print(name[-6])
IndexError: string index out of range
```

# Substrings/Slicing/Splicing?

- In addition to the index (let's call it the "start index"), you can also specify an "end index" in order to get all characters of the string *from* the start index *up to but excluding* the end index.

```
>>> name = "Eric"
>>> print(name[0:2])
Er
>>> print(name[1:3])
ri
>>> print(name[1:4])
ric
>>> print(name[1:])
ric
>>> print(name[2:])
ic
>>> print(name[:2])
Er
>>> print(name[:3])
Eri
>>> print(name[:])
Eric
```

- We would say that strings like "Er", "ri", and "ric" are **substrings** of "Eric". Some texts/sources call them **string slices** and say that the above is **slicing**. Some other sources call the above **splicing** instead.

# Practice

- Fill in the `print` statements below:

```
>>> letters = "abcdefgh"
>>> print(...)
cde
>>> print(...)
efgh
>>> print(...)
bcdef
>>> print(...)
abc
```

# Practice

- Fill in the `print` statements below:

```
>>> letters = "abcdefgh"
>>> print(letters[2:5])
cde
>>> print(...)
efgh
>>> print(...)
bcdef
>>> print(...)
abc
```

# Practice

- Fill in the `print` statements below:

```
>>> letters = "abcdefgh"
>>> print(letters[2:5])
cde
>>> print(letters[4:])
efgh
>>> print(...)
bcdef
>>> print(...)
abc
```

# Practice

- Fill in the `print` statements below:

```
>>> letters = "abcdefgh"
>>> print(letters[2:5])
cde
>>> print(letters[4:])
efgh
>>> print(letters[1:6])
bcdef
>>> print(...)
abc
```

# Practice

- Fill in the `print` statements below:

```
>>> letters = "abcdefgh"
>>> print(letters[2:5])
cde
>>> print(letters[4:])
efgh
>>> print(letters[1:6])
bcdef
>>> print(letters[:3])
abc
```

# Slicing with a Step

- You can also add a *third* integer to specify the "step" of the slicing.

```
>>> letters = "abcdefghijklmnop"
>>> print(letters[1:8])
bcdefgh
>>> print(letters[1:8:2])
bdfh
>>> print(letters[1:8:3])
beh
```

| **Index** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **...** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Char | a | **b** | c | d | **e** | f | g | **h** | i | j | ... |

- By default (i.e. if not specified), the "step" is 1, so the following statements are identical:

```
>>> print(letters[1:8])
bcdefgh
>>> print(letters[1:8:1])
bcdefgh
```

# Practice

- Fill in the `print` statements.

```
>>> chars = "xafrtqrw"
>>> print(chars[...])
xf
>>> print(chars[...])
rr
>>> print(chars[...])
xt
```

# Practice

- Fill in the `print` statements.

```
>>> chars = "xafrtqrw"
>>> print(chars[:4:2])
xf
>>> print(chars[...])
rr
>>> print(chars[...])
xt
```

# Practice

- Fill in the `print` statements.

```
>>> chars = "xafrtqrw"
>>> print(chars[:4:2])
xf
>>> print(chars[3:7:3])
rr
>>> print(chars[...])
xt
```

# Practice

- Fill in the `print` statements.

```
>>> chars = "xafrtqrw"
>>> print(chars[:4:2])
xf
>>> print(chars[3:7:3])
rr
>>> print(chars[::4])
xt
```

# Slicing with a Backwards Step

- You can provide a negative step as well (to print a substring in reverse), so long as the first index exceeds the second index.

```
>>> chars = "abcdefghijklmn"
>>> chars[11:2:-2]
'ljhfd'
```

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **...** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | **d** | e | **f** | g | **h** | i | **j** | k | **l** | m | ... |

# Slicing Can Go Out of Bounds

```
>>> word = "awesome"
>>> len(word)
7
>>> word[4:18]
'ome'
```

# Example: Count Letter 'a'

## Prompt

- While going through the `for` loop lecture slides, we wrote a program that asks the user for a string and prints the number of occurrences of the lowercase "a" letter.
  - For example, if the user enters "banana", the program should print 3.
  - **Prompt #1**: Rewrite this program so that it uses a `for ... in range(...)` statement.
  - **Prompt #2**: Then, rewrite the program so that it uses a `while` loop.

# Example: Count Letter 'a'

## Solution #1

```
s = input("Enter: ")
counter = 0
# explicit version: for i in range(0, len(s), 1):
for i in range(len(s)):
    if s[i] == "a":
        counter += 1
print("The letter \"a\" occurs {} times in {}.".format(
      counter, s))
```

## Solution #2

```
s = input("Enter: ")
counter = 0
i = 0
while i < len(s):
    if s[i] == "a":
        counter += 1
    i += 1
print("The letter \"a\" occurs {} times in {}.".format(
      counter, s))
```

# Traversing a String

- As shown in the previous example, we now have two ways of traversing a string, as compared below:

```
chars = "abcde"
for c in chars:
    print(c)
```

Output:

```
a
b
c
d
e
```

```
chars = "abcde"
for i in range(len(chars)):
    print(chars[i])
```

Output:

```
a
b
c
d
e
```

- Both are equally useful, but in some cases, one is arguably more convenient than the other.

# Example: Find a Character

## Prompt

- **Prompt #1**: Write a program that prompts the user to enter a string and a character. The program should then print the first index at which the given character occurs in the given string. If the character was not found, then the program should let the user know. Use a range-based `for` loop (i.e. `for ... in range(...)`).
- **Prompt #2**: Redo the above, but with a non-range-based `for` loop.
- **Prompt #3**: Redo the above, but use a `while` loop.

# Example: Find a Character

## Solution #1: `range()`-based `for` Loop

```
s = input("Enter string: ")
target = input("Enter target: ")
if len(target) == 1:
    found = False
    for i in range(len(s)):
        if s[i] == target:
            print("Target first occurs at index {}.".format(i))
            found = True
            break
    # Print message if target is not found.
    if not found:
        print("Could not find target.")
else:
    print("Target character was not one character.")
```

# Example: Find a Character

## Solution #2: Non-`range()`-based `for` Loop

```
s = input("Enter string: ")
target = input("Enter target: ")
if len(target) == 1:
    found = False
    i = 0
    for c in s:
        if c == target:
            print("Target first occurs at index {}.".format(i))
            found = True
            break
        i += 1
    # Print message if target is not found.
    if not found:
        print("Could not find target.")
else:
    print("Target character was not one character.")
```

# Example: Find a Character

## Solution #3: `while` Loop

```
s = input("Enter string: ")
target = input("Enter target: ")
if len(target) == 1:
    i = 0
    while i < len(s):
        if s[i] == target:
            print("Target first occurs at index {}.".format(i))
            break
        i += 1
    # Print message if target is not found.
    if i == len(s):
        print("Could not find target.")
else:
    print("Target character was not one character.")
```

# Example: Find the Last

## Prompt

- Repeat the previous program (all three versions) and print the index of the *last* index (at which the given character occurs in the given string) instead of the first.

## Solution #1: Forward Iteration with `range()`-based `for` Loop

```
s = input("Enter string: ")
target = input("Enter target: ")
if len(target) == 1:
    # Last index at which the target was found.
    target_last_found = -1
    for i in range(len(s)):
        if s[i] == target:
            target_last_found = i
    if target_last_found == -1:
        print("Could not find target.")
    else:
        print("Target last occurs at index {}.".format(
              target_last_found))
else:
    print("Target character was not one character.")
```

# Example: Find the Last

## Solution #2: Backwards Iteration with `range()`-based `for` Loop

```
s = input("Enter string: ")
target = input("Enter target: ")
if len(target) == 1:
    found = False
    # Example: if s = "Hello" (length 5), then we want
    # i to take on these values: 4, 3, 2, 1, 0
    for i in range(len(s) - 1, -1, -1):
        if s[i] == target:
            print("Target last occurs at index {}.".format(i))
            found = True
            break
    # Print message if target is not found.
    if not found:
        print("Could not find target.")
else:
    print("Target character was not one character.")
```

# Example: Find the Last

## Solution #3: Non-`range()`-based `for` Loop

```
s = input("Enter string: ")
target = input("Enter target: ")
if len(target) == 1:
    found = False
    i = len(s) - 1
    for c in s[::-1]:
        if c == target:
            print("Target last occurs at index {}.".format(i))
            found = True
            break
        i -= 1
    # Print message if target is not found.
    if not found:
        print("Could not find target.")
else:
    print("Target character was not one character.")
```

# The `in` Operator

- Use `in` to check if a string can be found in another string.

```
>>> "a" in "banana"
True
>>> "ana" in "banana"
True
>>> "ax" in "banana"
False
```

- As shown above, the `in` operator can tell you if a string *exists* in another string, but it cannot tell you *where* the string exists in the other string.

```
>>> # Okay, I'll take your word for it.
>>> "rx" in "kja;slfk$#jio!@wejlajoupqvprxqpi$uvald$@"
True
```

# Immutability

- This is prohibited:

```
>>> chars = "abcde"
>>> chars[2] = "X"
Traceback (most recent call last):
  File "<pyshell#149>", line 1, in <module>
    chars[2] = "X"
TypeError: 'str' object does not support item assignment
```

- This is because strings are **immutable**; you cannot *change* them.

# Immutability: Example of Workaround

- How can we *change* the third character to be "X"?
  - Strictly speaking, we *cannot*; we must instead create a new string. Fill in the blanks (`...`) below:

```
>>> new_chars = chars[...] + "X" + chars[...]
>>> print(new_chars)
abXde
```

# Immutability: Example of Workaround

```
>>> new_chars = chars[:2] + "X" + chars[3:]
>>> print(new_chars)
abXde
```

- We'll get to practical examples of this in the next set of slides.

# Practice

- What is the output of each of the following operations?

```
>>> "xyz" * 2 + " " + "x" * 3
???
>>> chars = "abcdef"
>>> chars[1:4:2]
???
>>> chars[3] = "x"
???
```

# String Methods

- If you have to use a dot between a string a function to use that function, then that function is a **method**. Since it is a method to be used on a string, I call it a **string method**.

```
>>> "AdXfR".lower()
'adxfr'
>>> "at2".isalpha()
False
>>> "at".isalpha()
True
>>> "at2".isalnum()
True
>>> "852".isdigit()
True
>>> "85 2".isdigit()
False
```

- Note that `len()` is not a string method; it does not follow the definition given above.

```
>>> len("85 2")
4
```

# String Methods

- Note that string methods work with string variables, not just string literals. None of them change the variable[1].

```
>>> chars = "abcde"
>>> print(chars.upper())
ABCDE
>>> print(chars)
abcde
```

- For a whole collection of string methods: https://www.w3schools.com/python/python_ref_string.asp

1. This is not true of methods in general, as we will learn when we talk about classes.

# Example: New Password

## Prompt

- Write a program that prompts the user to enter a new password. The program should then tell the user if this password fulfills the following requirements:
  - At least one capital letter.
  - At least two digits.
  - At least one non-alphanumeric character (e.g. a dollar sign).
  - At least eight letters.
- *Hint*: Implement each check of the requirements one at a time.

# Example: New Password

## Solution

```
password = input("Enter new password: ")

# Check if @password has capital letter.
has_capital = False
for c in password:
    if c.isupper():
        has_capital = True
        break  # just to save time (optional)

# Check if @password has at least two digits.
num_digits = 0
for c in password:
    if c.isdigit():
        num_digits += 1

# Check if @password has at least one non-alphanumeric character.
has_non_alnum = False
for c in password:
    if not c.isalnum():
        has_non_alnum = True
        break  # just to save time (optional)

# Check if @password has at least eight letters.
has_eight = False
if len(password) >= 8:
    has_eight = True
# has_eight = len(password) >= 8

if has_capital and num_digits >= 2 and has_non_alnum and has_eight:
    print("Valid password!")
else:
    print("Invalid password...")
```

# Appendix: ASCII

- As you will learn in ECS 50, your computer stores all data as (binary) numbers. This means that each character must have a numerical representation. The ASCII table provides numerical representations of 128 different characters, as you can see in the image below[1]. Only look at the Dec and Char/Chr columns; the Hx and Oct columns will not make sense until ECS 50.

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

Source: www.LookupTables.com

1. Source: http://www.asciitable.com/

# Appendix: ASCII

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

Source: www.LookupTables.com

- For example, according to the above:
  - "A" has an ASCII value of 65
  - "H" has an ASCII value of 72
  - "a" has an ASCII value of 97
  - " " (i.e. "Space") has an ASCII value of 32
  - "$" has an ASCII value of 36
- Now does it make sense why uppercase letters come before lowercase letters in the programmer alphabet?

# Appendix: ASCII

- Use the `ord` and `chr` built-in functions to convert a character to its ASCII value or vice-versa:

```
>>> ord("A")
65
>>> chr(65)
'A'
```

# Appendix: Unicode

- Unicode is an even bigger set of characters, contaning numerical values for 137,994 characters currently[1]. This allows Unicode to have numerical representations for all kinds of characters, from symbols from other languages to emojis. New characters are added as time goes along.

1. Source: https://en.wikipedia.org/wiki/Unicode