15-110 Principles of Computing – F19

Lecture 8:
Strings 2

Teacher:
Gianni A. Di Caro

So far about Python …

- Basic elements of a program:
  - Literal objects
  - Variables objects
  - Function objects
  - Commands
  - Expressions
  - Operators
- Utility functions (built-in):
  - `print(arg1, arg2, …)`
  - `type(obj)`
  - `id(obj)`
  - `int(obj)`
  - `float(obj)`
  - `bool(obj)`
  - `str(obj)`
  - `input(msg)`
  - `len(non_scalar_obj)`

- Object properties
  - Literal vs. Variable
  - Type
  - Scalar vs. Non-scalar
  - Immutable vs. Mutable
- Conditional flow control
  - ```
    if cond_true:
        do_something
    ```
  - ```
    if cond_true:
        do_something
    else:
        do_something_else
    ```
  - ```
    if cond1_true:
        do_something_1
    elif cond2_true:
        do_something_2
    else:
        do_something_else
    ```

- Data types:
  - `int`
  - `float`
  - `bool`
  - `str`
  - `None`
- Relational operators
  - `>`
  - `<`
  - `>=`
  - `<=`
  - `==`
  - `!=`
- Logical operators
  - `and`
  - `or`
  - `not`

- Operators:
  - `=`
  - `+`
  - `+=`
  - `-`
  - `/`
  - `*`
  - `*=`
  - `//`
  - `%`
  - `**`
  - `[ ]`
  - `[ : ]`
  - `[ : : ]`
    ```

Immutability of string objects

- Strings are **immutable** object types: we can't change the value of a string literal / variable!
  - Once created, the string keeps its value for its entire lifetime

```
s = 'abcd'
s[3] = 'z'
```

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

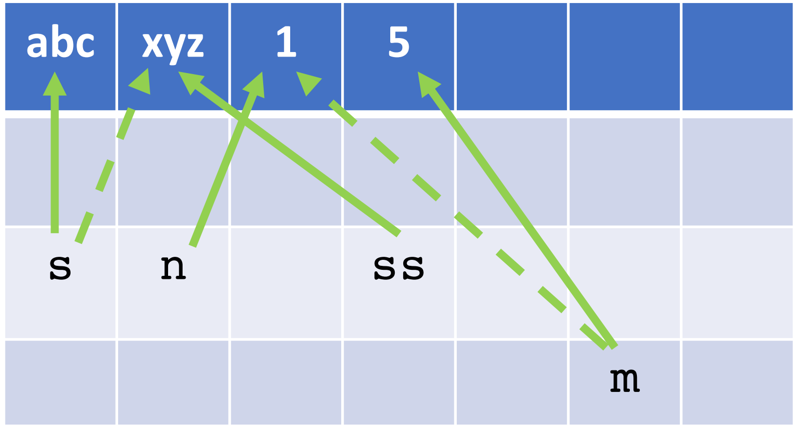
- We can "change" the value of a string variable by *reassigning* its value, which amounts to create a new string variable

```
s = 'abcd'
print(id(s))
s = 'abcz'
print(id(s))
```

```
4813142648
4814169064
```

Immutability of strings

```
s = 'abc'
ss = 'xyz'
n = 1
m = 5
print( id(s), id(ss),
       id(n), id(m))
```




```
4652072720 4327641640 4307354688 4307354816
```

```
s = 'xyz'
ss = 'xyz'
n = 1
m = n
print( id(s), id(ss), id(n),
       id(m), id(1), id('xyz'))
```

```
4652072720 4652072720 4307354688 4307354688
```

```
4307354688 4652072720
```

Built-in String *Methods*

- **Functions**: callable procedures that can be invoked to perform specific tasks (can take generic arguments)
- **Method**: a specialized type of callable procedure that is tightly associated with an object. Like a function, a method is called to perform a precise task, but it is invoked on a specific object and has knowledge of its target object during execution
  → Will understand this more when study *class* objects
- **Dot notation** for invoking a method on an object: `obj.method_name(arguments)`

Obj dot Method name Args go here

```
s = 'Hello Joe'
ss = s.lower()
print(ss)
```

hello joe

```
s = 'Hello Joe'
ss = s.upper ()
print(ss)
```

HELLO JOE

Let’s look at methods built-in with the string (class) type, and let `s` be a string variable

Full List with examples! https://www.w3schools.com/python/python_ref_string.asp

Built-in String Methods: Case Conversion

- `s.lower()`: returns a *copy* of `s` with all alphabetic characters converted to lowercase
- `s.upper()`: returns a copy of `s` with all alphabetic characters converted to uppercase
- `s.capitalize()`: returns a copy of `s` with the first character converted to uppercase and all other characters converted to lowercase
- `s.swapcase()`: returns a copy of `s` with uppercase alphabetic characters converted to lowercase and vice versa. Non-alphabetic characters are unchanged.
- `s.title()`: returns a copy of `s` in which the first letter of each word (separated by spaces) is converted to uppercase and remaining letters are lowercase

Methods that do not modify the calling object

returns a copy of s → These are methods that **do *not* modify the original object,** but rather return a *copy of it with the specified changes*

```
s = ‘Hello Joe'
s_new = s.lower()
print("s:", s, '-', "reference address: ", id(s), '\n'
      "s_new:", s_new, '-', "reference address: ", id(s_new))
```

`s.upper()` → Nothing happens, the copy of `s` is generated but not being used

Why?

Because strings are immutable data types, such that we can really modify a string object!

Built-in String Methods: Count, Find, and Replace

- `s.count(<sub>):` returns the number of non-overlapping occurrences of substring <sub> in s
  - s = “moo ooh pooh”
  - s.count(“oo”) → 3
  - “moo ooh pooh”.count(“oo”)
- `s.count(<sub>, <start>, <end>):` returns the number of non-overlapping occurrences of substring <sub> in the s slice defined by <start> and <end>
  - s = “moo ooh pooh”
  - s.count(“oo”, 3, len(s)) → 2

Built-in String Methods: Count, Find, and Replace

- `s.endswith(<suffix>):` returns True if s ends with the specified <suffix>, and False otherwise
  - s = “crazy bar”
  - s.endswith(“bar”) → True
- `s.endswith(<suffix>, <start>, <end>):` as above, but now the comparison is restricted to the substring indicated by <start> and <end>
  - s = “crazy bar”
  - s.endswith(“bar”, 0, 5) → False
- `s.startswith(<prefix>):` analogous to `endswith(),` but checking if the string begins with a given substring

Built-in String Methods: Count, Find, and Replace

- `s.find(<sub>):` returns the lowest index in s where the substring <sub> is found, -1 is returned if the substring is not found
  - s = “crazy bar bar”
  - s.find (“bar”) → 6
  - s.find(“star”) → -1
- `s.find(<sub>, <start>, <end>):` as above, but now the search is restricted to the substring indicated by <start> and <end>
  - s = “crazy bar bar”
  - s.find(“bar”, 7, 13) → 10

Built-in String Methods: Count, Find, and Replace

- `s.rfind(<sub>):` searches s starting from the end, such that it returns the highest index in s where the substring <sub> is found, -1 is returned if the substring is not found

  - s = "crazy bar bar"
  - s.rfind ("bar") → 10
  - s.find("bar") → 6

- `s.rfind(<sub>, <start>, <end>):` as above, but now the search is restricted to the substring indicated by <start> and <end>

  - s = "crazy bar bar"
  - s.rfind("bar", 0, 10) → 6

Built-in String Methods: Count, Find, and Replace

- `s.replace(<old>, <new>):` returns a *copy* of s with all occurrences of substring <old> replaced by new. If there are no occurrence of <old>, the copy is identical to the original (but it's still a different object)
  - s = "one step, two steps, three steps"
  - s.replace ("step", "stop") → "one stop, two stops, three stops"
- `s.replace(<old>, <new>, <max>):` as above, but now the number of replacements is limited to the <max> value
  - s = "one step, two steps, three steps"
  - s.replace("step", "stop", 2) → "one stop, two stops, three steps"

Built-in String Methods: String formatting

- s.center(<width>[, <fill>])
- s.expandtabs(tabsize=8)
- s.ljust(<width>[, <fill>])
- s.lstrip([<chars>]): It removes all leading whitespaces of a string and can also be used to remove a particular character from leading
- s.rjust(<width>[, <fill>])
- s.rstrip([<chars>]): >]): It removes all trailing whitespaces of a string and can also be used to remove a particular character from trailing
- s.strip([<chars>]): It removes all leading and trailing whitespaces of a string and can also be used to remove a particular character from both leading and trailing
- s.zfill(<width>)

Built-in String Methods: Character classification

- `s.isalpha()`: `True` if all characters in `s` are alphabetic letters, `False` otherwise
- `s.isalnum()`: `True` if all characters in `s` are either alphabetic letters or numeric digits, `False` otherwise
- `s.isdigit()`: `True` if all characters in `s` are numeric digits, `False` otherwise
- `s.isidentifier()`: `True` if the string `s` could be used as identifier (variable, function or class name), `False` otherwise
- `s.islower()`: `True` if all characters in `s` are lower case, `False` otherwise
- `s.isupper()`: `True` if all characters in `s` are lower case, `False` otherwise
- `s.isprintable()`: `True` if all characters in `s` are printable, `False` otherwise
- `s.isspace()`: `True` if all characters in `s` are white spaces, `False` otherwise
- `s.istitle()`: `True` if the first character in `s` is upper case and all the others are lower case, `False` otherwise

String formatting using escape sequences

- print("He said, "What's there?" ") → SyntaxError: Invalid syntax
- print('He said, "What's there?" ') → SyntaxError: Invalid syntax
- Alternative way: Use of **Escape sequences**
  - An escape sequence **starts with a backslash ** such that what follows <u>is interpreted differently from usual</u> (it is *protected*)
  - print("He said, \"What's there? \" ") → Ok
  - print('He said, "What\'s there?" ') → Ok
- \n : **new line feed** is inserted print(" Hello!\nThis goes on a new line ")
- \t : **tabular space** is inserted print(" Hello!\t\tThis gets two tab spaces ")
- \\ : this allows to write file/folder **paths in windows** print("C:\\Python64\\Lib")
- \a : this **rings a bell**! print(" This rings a bell\a")

String formatting using escape sequences

Escape Sequence	Description
\newline	Backslash and newline ignored
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\f	ASCII Formfeed
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab
\ooo	Character with octal value ooo
\xHH	Character with hexadecimal value HH

ASCII encoding for characters

- **Encoding:** Character → Integer number → Binary representation
- ASCII (American Standard Code for Information Interchange) standard code, defined in 1968 (and extended later on), assigns a <u>numeric code</u> (that can be hold in **8 bits = 1 byte**) to a subset of standard characters
- **1 byte: basic unit of storage in computer memory!**

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Unicode encoding

- **Encoding:** Character (e.g., c)→ Integer number (143) → Binary representation (10001111)
- Developed in recent times to address the widespread use of computers in different countries using different symbols in their alphabet
- Different **Unicode codes** are around, using encoding larger (and more complex) than the ASCII's 8 bits, allowing to index code points (characters) large enough, to represent virtually any language around
- The most commonly used Unicode encoding is **UTF-8**, that is fairly compact and includes ASCII codes
- Your Spider makes use of UTF-8!

Conversions between different numeric bases (optional)

- Let's consider an **integer number x** with $n = 5$ digits, e.g., $x = 64523$
- This is a base 10 $(b = 10)$ representation of the number, using digits from 0 to 9
  - $x = 6 \cdot 10^4 + 4 \cdot 10^3 + 5 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 64{,}523$

Position	4	3	2	1	0
Exponent	10^4	10^3	10^2	10^1	10^0
Value	10,000	1,000	100	10	1
Digits	x_4	x_3	x_2	x_1	x_0

- Let's consider now a **binary number x** with $n = 5$ digits, e.g., $x = 11001$
- This is a base 2 $(b = 2)$ representation of the number, using digits 0 and 1
- What is the integer value of the number x?
  - $x = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 25$

Position	4	3	2	1	0
Exponent	2^4	2^3	2^2	2^1	2^0
Value	16	8	4	2	1
Digits	x_4	x_3	x_2	x_1	x_0

Binary	Octal	Decimal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
Base-2	*Base-8*	*Base-10*	*Base-16*

From binary to decimal representation (optional)

Position	7	6	5	4	3	2	1	0
Exponent	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	128	64	32	16	8	4	2	1
Digits	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0

Most Significant Bit
(MSB)

Least Significant Bit
(LSB)

$$x = x_7 \cdot 2^7 + x_6 \cdot 2^6 + x_5 \cdot 2^5 + x_4 \cdot 2^4 + x_3 \cdot 2^3 + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

- How many *unsigned* integer numbers can be represented with 8 bits? → 256
- How many *signed* integer numbers can be represented with 8 bits? → 128
  (we need to reserve one bit for $+$ / $-$)
- Internal *non-scalar* representation of numbers

Bits (optional)

- **One bit** (that can take on two values, 0 or 1)
  - We can represent 2 integer numbers: 0 1
  - The max value of an integer that we can represent with 1 bit: 1

- **Two bits**
  - We can represent 4 integer numbers: 00 01 10 11, from 0 to 3
  - The max value of an integer that we can represent with 2 bits: 3 (obtained from $2^2 - 1$)

- **Three bits**
  - We can represent 8 integer numbers: 000 010 100 110 011 101 001 111, from 0 to 7
  - The max value of an integer that we can represent with 2 bits: 7 (obtained from $2^3 - 1$)

....

- **8 bits = 1 byte**
  - We can represent 256 (unsigned) integer numbers: from 0 to 255
  - The max value of an integer that we can represent with 8 bits: 255 (obtained from $2^8 - 1$)

Conversion from base 10 to base 2 representations (optional)

Position	7	6	5	4	3	2	1	0
Exponent	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	128	64	32	16	8	4	2	1
Digits	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0

MSB LSB

- From base 10 to base 2?
- Keep dividing by 2 and storing the remainder
- → Modulo operation!!!

Number	294		
divide by 2			
result	147	remainder	0 (LSB)
divide by 2			
result	73	remainder	1
divide by 2			
result	36	remainder	1
divide by 2			
result	18	remainder	0
divide by 2			
result	9	remainder	0

Dividing each decimal number by “2” as shown will give a result plus a remainder.

If the decimal number being divided is even then the result will be whole and the remainder will be equal to “0”. If the decimal number is odd then the result will not divide completely and the remainder will be a “1”.

divide by 2			
result	4	remainder	1
divide by 2			
result	2	remainder	0
divide by 2			
result	1	remainder	0
divide by 2			
result	0	remainder	1 (MSB)

The binary result is obtained by placing all the remainders in order with the least significant bit (LSB) being at the top and the most significant bit (MSB) being at the bottom.

$$(294)_{10} = (100100110)_2$$