Chapter 8: Strings

Stephen Huang

March 23, 2023

UNIVERSITY of HOUSTON

Contents

1. Introduction to Characters and Strings
2. String Manipulations Methods
3. String Comparison
4. String Formatting
   - With C-Style
   - With String format() *
   - With String Template
   - With f-strings *

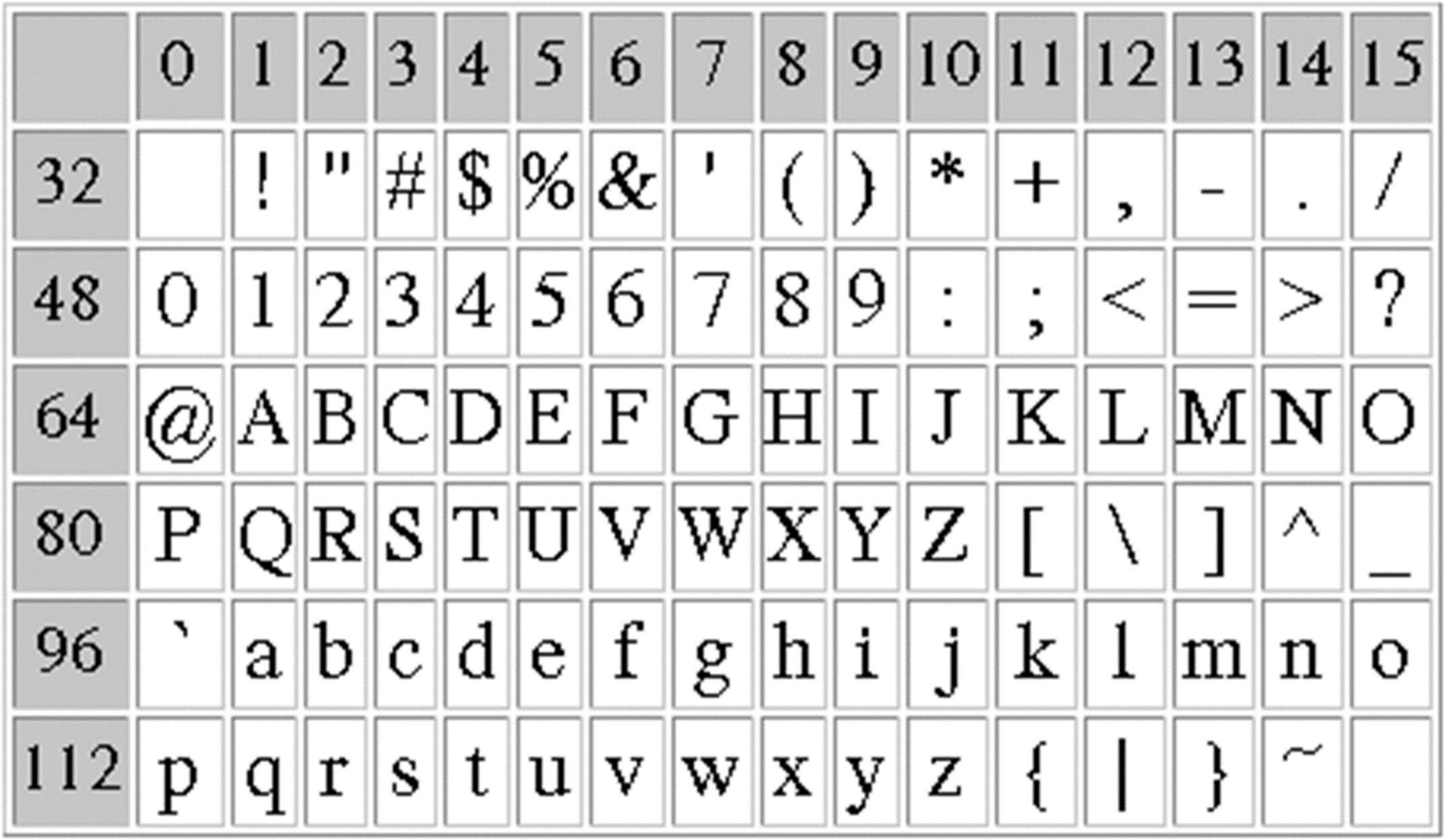
1. Introduction

- A string-type object is a sequence of characters.
- In Python, strings start and end with single- or double-quotes.
- Each string is stored in computer memory as a “special” list (array, vector) of characters.
- Python string variable consists of a pointer to the position in computer memory (the address) of the 0th byte.
- Every byte in your computer memory has a unique integer address.

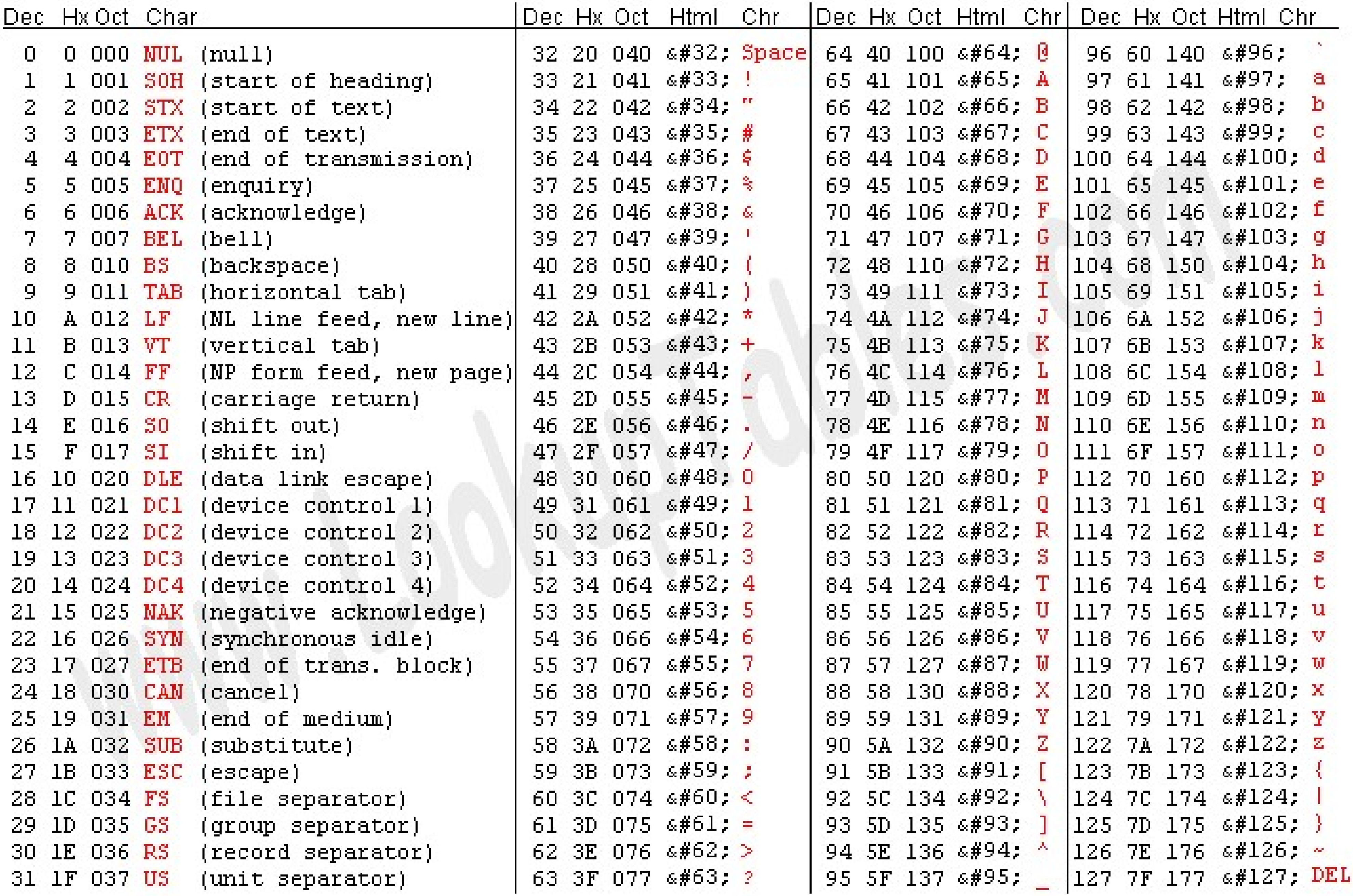
Character Encoding

- Two commonly used character encodings are ASCII (128 characters) and Unicode (1,114,112 characters).
- Fortunately, they share the same numerical to character values. ‘A’ is coded as 65 in both systems.
- No need to worry about too much.

Printable ASCII



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	



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

String

- In some other programming languages, strings are terminated by an extra special character which is not the case in Python.
- For example,
  - `"Test"` consists of only four characters.
  - `""` is an empty string.

Ordering

- Note that the order of the character codes is such that
  - '0' < '1' < ... < '9'
  - 'A' < 'B' < ... < 'Z'
  - 'a' < 'b' < ... < 'z'.
- There are no other characters in the three sequences above. They are consecutive.
- So, two letters will compare as expected if the two letters are both of the same cases.
  - For example, `'A'` < '`D`' and '`a`' < '`d`'.
  - However, '`D`' < '`a`' because all the uppercase letters have character codes less than the lowercase letters.

Ordering

- The letters do not compare correctly in alphabetical order if the letters are in different cases.
- It would be best to ensure the compared letters are in the same case.
  - Use string.lower(), string.upper()
- One can assign a value to a variable of type char, e.g., ch = 'A'.

Ordering

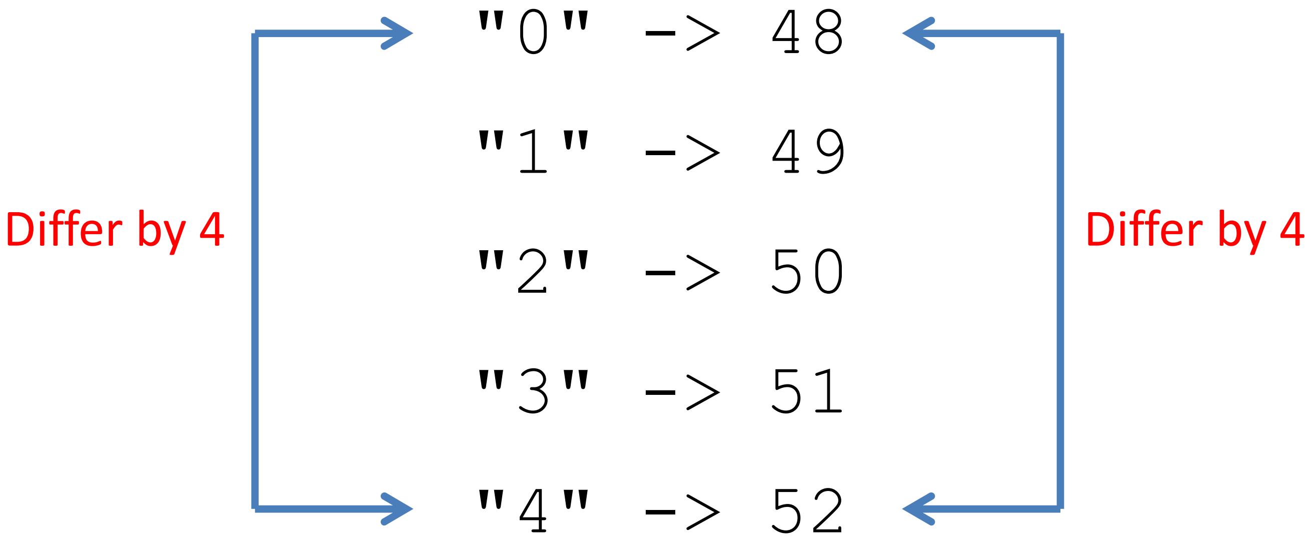
- To convert the character to the corresponding ASCII code (an ordinal number), one can use the ord() function.

```
ord('a') = 97
ord('A') = 65
```

- To convert an integer to an ASCII character: use the chr() function.

```
chr(65) = 'A'
```

Ordinal number of digits



Differ by 4

Differ by 4

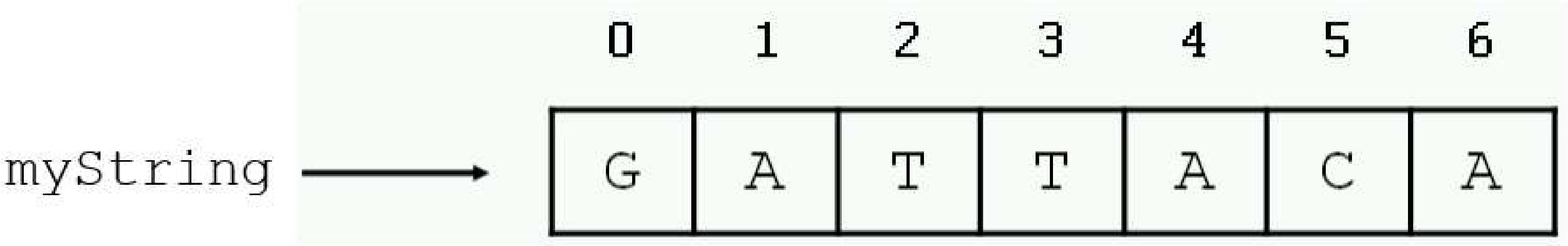
```
"0" -> 48
"1" -> 49
"2" -> 50
"3" -> 51
"4" -> 52
"5" -> 53
"6" -> 54
"7" -> 55
"8" -> 56
"9" -> 57
```

Example

```
def c2i(ch):
    return ord(ch)-ord('0')

def i2c(i):
    return chr(i+ord('0'))

ch = '7'
print(f'"{ch}" converts into {c2i(ch)}.')
i = 6
print(f'{i} converts into "{i2c(i)}".')
```

Accessing a single character

- `myString = “GATTACA”`

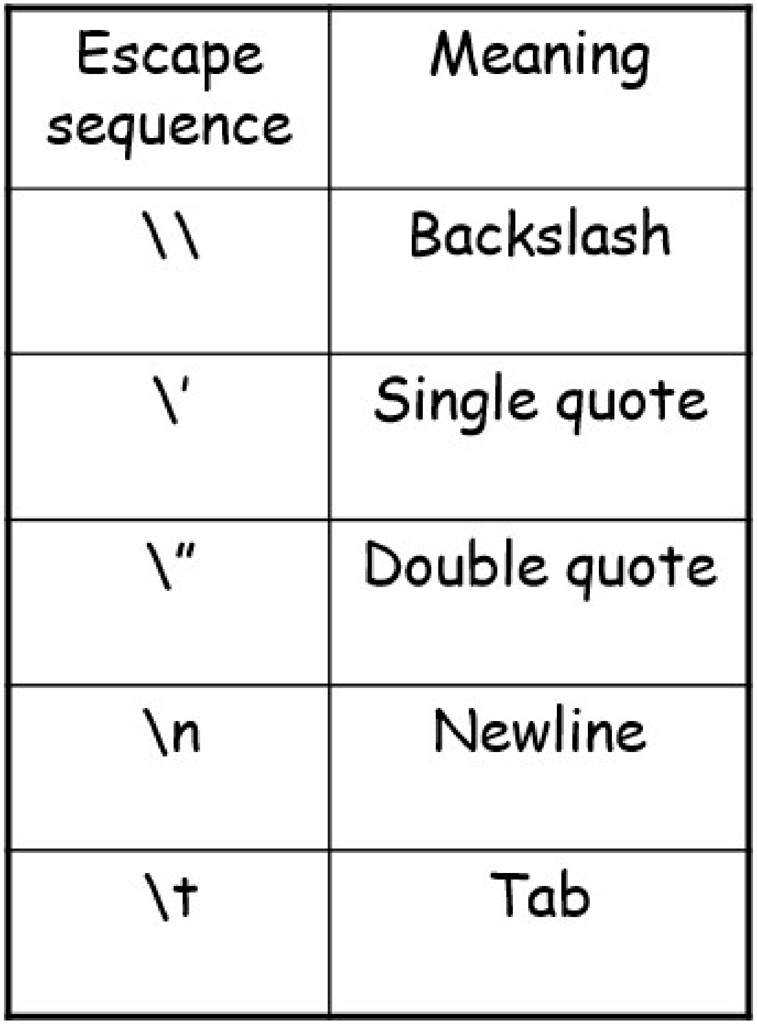


0	1	2	3	4	5	6
G	A	T	T	A	C	A

computer memory (7 bytes)

- You can access individual characters by using indices in square brackets.
  - myString[0], myString[2], myString[-1], but no myString[7]

Special Characters



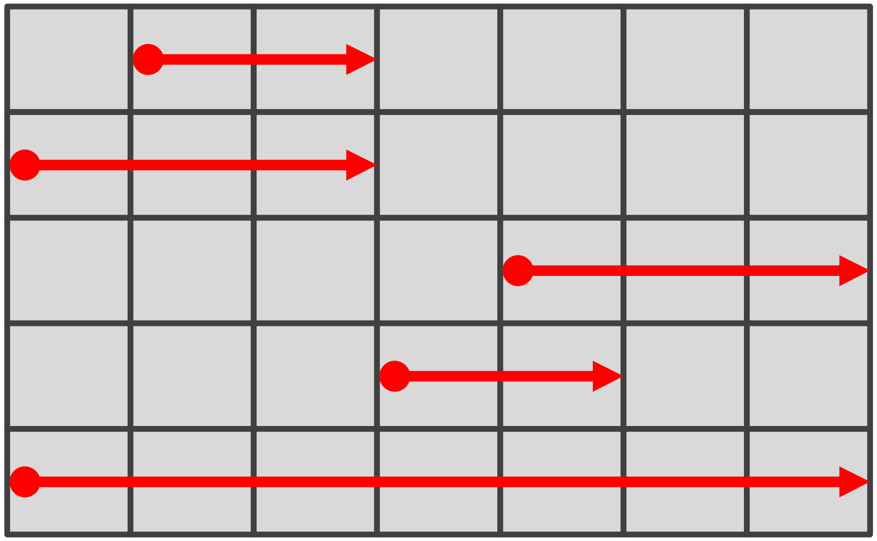
Escape sequence	Meaning
\\	Backslash
\'	Single quote
\"	Double quote
\n	Newline
\t	Tab

Slicing

```
str = “Houston”
```

0	1	2	3	4	5	6
H	o	u	s	t	o	n

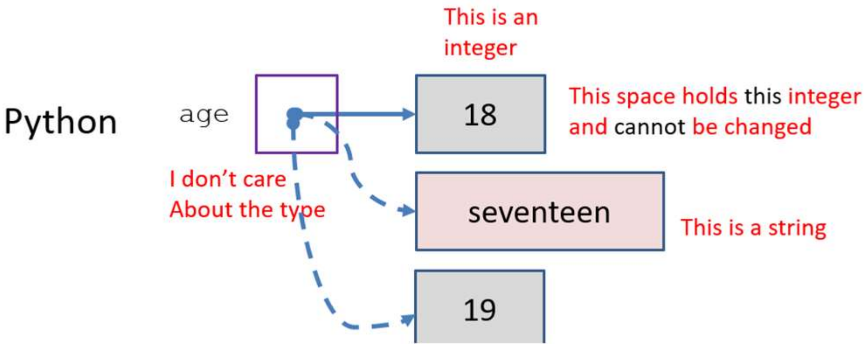
```
str[1:3]

str[:3]

str[4:]

str[3:5]

str[:]
```



Immutable

- Strings cannot be modified; instead, create a new string for the new value. (List is mutable.)

```
>>> greeting = 'Hello, world!'
>>> greeting[0]
'H'
>>> greeting[0]='J'
TypeError: 'str' object does not
support item assignment
```



Python

age

This is an integer

18

This space holds this integer and cannot be changed

I don't care About the type

seventeen

This is a string

19

Example

```
>>> greeting[1:]
'ello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
>>>
```

Search Example

```
def find(word, letter):
    index = 0
    while index<len(word):
        if word[index] == letter:
            return index
        index = index+1
    return None
```

Search Example

```python
def find(word, letter):
    for i, ch in enumerate(word):
        if ch == letter:
            return i
    return None
```

2. String Manipulations

- Length
- Concatenation
- Repeat
- Substring test (IN)

```
str = “Houston”
len(str)
str + str
“UH” * 3
“Hou” in “Houston”
“hou” in str
```

String Methods

- In Python, a method is a function defined with respect to a particular object.
- The syntax is:

object.method (arguments)

```
>>> dna = "ACGT"
>>> dna.find("T")
3
```

the first position where “T” appears

String Operations

- S = "AATTGG
- s1 + s2
- s2 * 3
- s2[i]
- s2[x:y]
- len(S)
- int(S)
- float(S)

String Methods

- `S.upper()`
- `S.lower()`
- `S.count(substring)`
- `S.replace(old,new)`
- `S.find(substring)`
- `S.startswith(substring)`
- `S.endswith(substring)`

Replace

- The method `replace(old, new, max)` returns a copy of the string in which the occurrences of `old` have been replaced with `new`, optionally limiting the number of replacements to the `max`.

```
str = "this is string ..wow!!! this is string"
print(str.replace("is", "was"))
print(str.replace(" is ", " was "))
print(str.replace("is", "was", 3))
```

```
               thwas was string ..wow!!! thwas was string
               this was string ..wow!!! this was string
               thwas was string ..wow!!! thwas is string
```

Testing

- `word.isalnum()` #check if all char are alphanumeric
- `word.isalpha()` #check if all char in the string are alphabetic
- `word.isdigit()` #test if string contains digits
- `word.isupper()` #test if string contains upper case
- `word.islower()` #test if string contains lower case
- `word.isspace()` #test if string contains spaces
- `word.endswith('d')` #test if string endswith a d
- `word.startswith('H')` #test if string startswith H

3. String Comparison

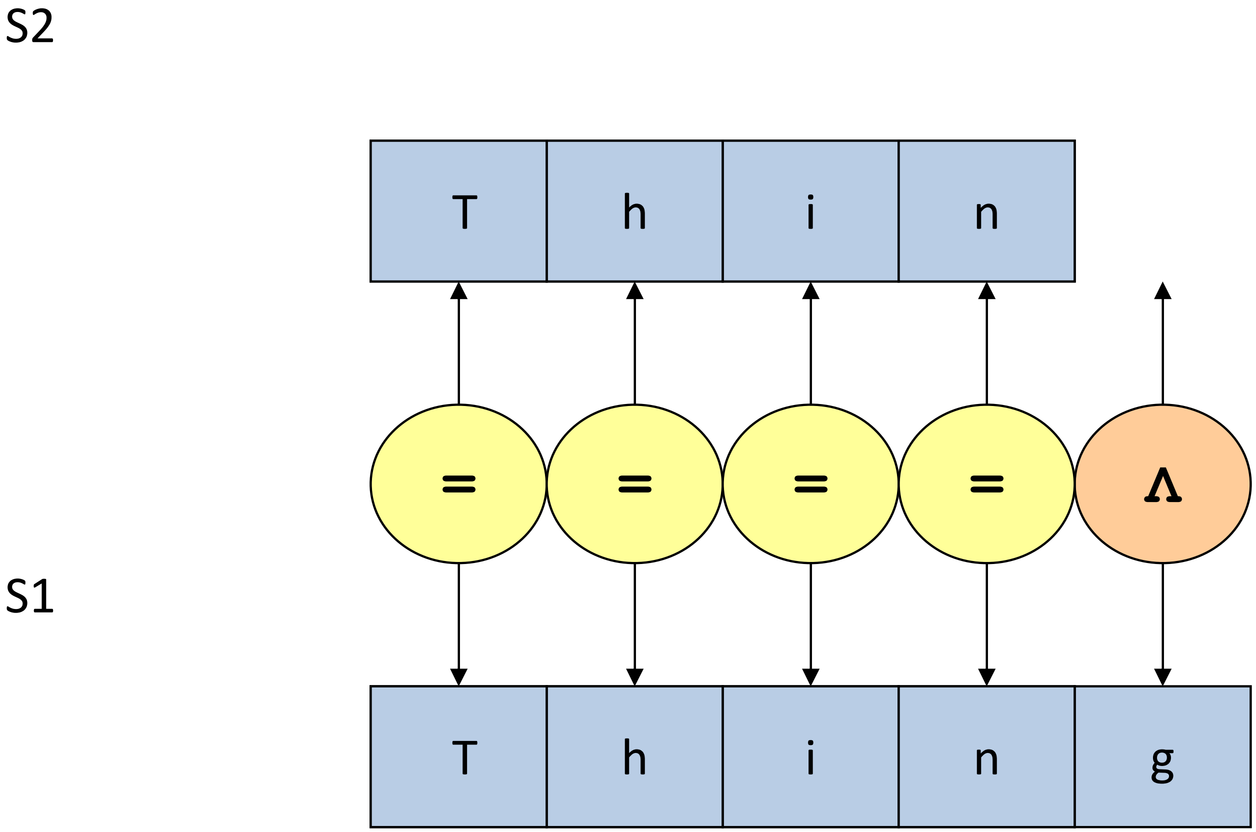
- You can compare two strings using the relational operators (==, !=, <, <=, >, >=).
- Relational operations help put words in alphabetical order.
- Note that upper-case letters come before lower-case letters in the ASCII table. We're not ordering alphabetically but ASCII-betically.
- A common way to address this problem is to convert strings to a standard format, such as all lowercase, before comparing.

Comparison

```
def swap(w1, w2):
    if w1 > w2:
        w1, w2 = w2, w1
    return w1, w2

w1 = 'pear'
w2 = 'apple'
w3 = 'Apple'
w1, w2 = swap (w1, w2)
print(w1, w2)
w1, w3 = swap(w1, w3)
print(w1,w3)
```

```
apple pear
Apple apple
```

String Comparison



Example

- We try to get a “clean” string from the user input in this example.
- A clean string is one without extra spaces separating two words. In other words, we will keep only one space between words and remove the additional spaces.

Example

```
def getStrClean():
    str = input("Enter a string: ")
    clean, space = '', False
    for ch in str:
        if ch==' ':
            if not space:
                clean = clean+ch
            space = True
        else:
            clean = clean+ch
            space = False
    return(clean)
print('[', getStrClean(),']', sep='')
```

Example

- There is one minor problem with the program.
  - See if you can find it.
  - How to fix it?
- This can be done quickly with a string method. See the explanation of these methods later.

```
clean = ' '.join(str.split())
```

String Processing

- `str.strip([chars])`
  - Chars: The characters to be removed from the beginning or end of the string.
  - This method returns a copy of the string in which all “chars” have been stripped from the string’s beginning and end.

```
str = "0000this is a string
example....wow!!!000";
Print(str.strip('0'))
```

String Processing

- `str.split([chars])`
  - It splits a string and adds the data to a list using a predefined separator string.
  - The most common separator is space.
  - If no separator is defined in the parameter, whitespace will be the default. In this case, all whitespaces will be removed.

`Str-split.py`

Join

- The reverse of the `split` is a `join`.
- If you have to join a list of words so that a space separates the words, how do you do it?
- Not that easy if you don’t want a space at the end.

```
' '.join(words)
```

4. String Formatting

- To produce readable output.
- We want to print many types of values (int, float, string, etc.), plus additional formatting information.
- Eventually, they are all combined into a string before printing.
- Two components:
  - Values (variables, literals, or expressions)
  - Formatting string (instruction on how to print)
- How do we mix the two?

Multiple Ways

- There are many ways to do so. Too many.
  - "Old Style" String Formatting (%-operator) before v2.6
  - "New Style" String formatting (str.format())
  - Template Strings (Standard Library)
  - String Interpolation (f-strings) after v3.6
- We will spend more time on the second and the fourth methods.
- Most of my notes use the f-string formatting.

Example

- We will use the same example for the comparison methods we discussed.
  - `name = 'John Smith'`
  - `acct_id = 12345678`
  - `balance = 123456.789`

Syntax Issues

- It is crucial to identify a place in a formatted string for values to be injected—a “placeholder.”
  - `"Name: {name}"`

- Sometimes, we need a symbol to separate a value with the formatting instructions.
  - `%[flags][width][.precision]type`

Formatting

- Formatting specifications include:
  - Types (of the value)
  - Width (of the value)
  - Precision (of a floating number)
  - Flags (various formatting specifications)

String Formatting Methods

- Since print() always print the content in a string of characters, it is possible to format it by calling string methods to change the string into a desired form first.
- Then, you can print the ‘formatted’ string.

String Methods

- There are several methods available for the string class for formatting the string. They are fairly limited.
  - `str.center(),`
  - `str.ljust(),`
  - `str.rjust(),`
  - `str.zfill()`

Examples

```
s = 'Python'
num = '12345’
# [] are added to show the white spaces

print('1. [', s, ']', sep='')
print('2. [', s.center(10), ']', sep='')
print('3. [', s.center(10,"*"), ']', sep='')
print('4. [', s.ljust(10), ']', sep='')
print('5. [', num.rjust(10, "*"), ']', sep='')
print('6. [', num.zfill(10), ']', sep='')
print('7. [', s.zfill(10), ']', sep='')
```

4.1 With C-Style Formatting

- This is the “old” style. Use it if you are using an older version of Python.
- Inherited from C-style printf() function.
- Given **`format%values`** (where the **`format`** is a string), % conversion specifications in format are replaced with zero or more values elements.
  - Example: `%5d, %6.2f, %s`

Example

```
name = 'John Smith'
acct_id = 12345678
balance = 123456.789


print("Name: %s  Id: %d  Balance: $%10.2f"
      % (name, acct_id, balance))
```

Use this format string

to format

these values

General Formatting

- Syntax: %[flags][width][.precision]type
  - Type
  - Width
  - Precision
  - Flags, options

- Example: %5d, %6.2f, %s

Alternative Way

```
name = 'John Smith'
acct_id = 12345678
balance = 123456.789
data = (name, acct_id, balance)
fmt_str = "Name: %s  Id: %d  Balance: $%9.2f"

print(fmt_str % data)
```

4.2 With String Format()

- The string class has a format() method.
- A format string contains code (fields to be replaced) embedded in the constant text.
- The template should be printed literally except for the format code (placeholder) to be filled in.
- The "placeholder" should be surrounded by curly braces {}.
- If a bracing character has to be printed, it has to be escaped by doubling it: {{ and }}.

Format()

- The curly braces and the "code" inside will be substituted with a formatted value from one of the arguments.
- Anything else not contained in curly braces will be printed without changes.
- There are two kinds of arguments for the .format() method:
  - positional arguments (0, 1, ...),
  - keyword arguments of the form name=value.

Example

```
fmt_str = "Name: {:s} Id: {:d} Balance:
${:9,.2f}"
```

By position

```
print(fmt_str.format(name,acct_id,balance))
```

```
print("Name: {:s} Id: {:d} Balance:
${:9,.2f}".format(name, acct_id, balance))
```

By index

```
print("Name: {0:s} Id: {1:d} Balance:
${2:9,.2f}".format(name, acct_id, balance))
```

By name

```
print("Name: {name:s} Id: {id:d} Balance:
${bal:9,.2f}".format(name=name, id=acct_id,
bal=balance))
```

Simplified Syntax

{[index]:[fill] [align] [sign] [width] [,] [.precision] [type]}

- Align: `< (default), >, =, ^`
- Fill: character to fill the space due to align. Default is space.
- Sign: `+, - (default), " "`
- Type: `d c e f s` etc.
- The ',' option signals the use of a comma for a thousands separator.

Signs

- `'+'`: indicates that a sign should be used for both positive and negative numbers.
- `'-'`: indicates that a sign should be used only for negative numbers (this is the default behavior).
- Space: indicates that a leading space should be used on positive numbers and a minus sign on negative numbers.

Commonly Used Types

- This is not a complete list.
  - d: signed integer decimal
  - e: floating point exponential format
  - f: floating point decimal format
  - c: single character
  - s: string
  - B: binary
  - o: octal
  - x: hex

Placeholder

- Placeholders can identify the value used for that placeholder by position (starting from 0) or by name.

Examples

```
template1="My name is {0} and I am {1} years old."
print(template1.format("Stephen", 59))
```

```
   My name is Stephen and I am 59 years old.
```

```
template2="My name is {} and I am {} years old."
print(template2.format("Stephen", 39))
```

```
   My name is Stephen and I am 39 years old.
```

```
template3="My name is {1} and I am {0} years old."
print(template3.format("Stephen", 29))
```

```
   My name is 29 and I am Stephen years old.
```

Examples

```
fmt_str1="[{:s}]  [{:s}]"
fmt_str2="[{:10s}]  [{:8s}]"
fmt_str3="[{0:^10s}]  [{1:>8s}]"

print(fmt_str1.format("Hello", "World."))
print(fmt_str2.format("Hello", "World."))
print(fmt_str3.format("Hello", "World."))
```

```
[Hello]  [World.]
[Hello     ]  [World.  ]
[  Hello   ]  [  World.]
```

Examples

```
fmt_str4="[{0:>10d}]  [{1:>15.3f}]"
fmt_str5="[{0:>10d}]  [{1:>+15.2f}]"
fmt_str6="[{0:0=10d}]  [{1:>15,.2f}]"

print(fmt_str4.format( 123, 123456.789))
print(fmt_str5.format(-123, 123456.789))
print(fmt_str6.format(-123, 123456.789))
```

```
[       123]  [     123456.789]
[      -123]  [     +123456.79]
[-000000123]  [     123,456.79]
```

4.3 With String Template

- Separating formatting (template string) from values.
- Probably the only time to use template strings is when you use formatted strings generated by others, such as program users.
- I don’t recommend this formatting method; you don’t need it now. That’s why I am showing a simple example here.

Examples

```
name = 'John Smith'
acct_id = 12345678
balance = 123456.789

from string import Template

t = Template("Name: $name Id: $id
Balance: $$$bal")
print(t.substitute(name=name, id=acct_id,
                   bal=balance))
```

4.4 With f-string

- Formatted string literals, also called format string or f-strings, is a feature added to Python 3.6.
  - Add an f or F before the quotes.
- Use curry braces {} as escape characters. Anything inside {} will be evaluated (replaced with their values
- Python f-strings provide a faster, more readable, more concise, and less error-prone way of formatting strings in Python.
- The f-strings have the `f` prefix and use `{}` brackets to evaluate values.

Why it is better?

- Python f-strings provide
  - A faster, more readable, more concise, and less error-prone way of formatting strings in Python.
  - The ability to print variable names with the value is great for debugging.
  - The ability to embed formatting operations into the modifiers.
  - Nested f-strings, conditional formatting, Lambda expression

From f-string to string

- Can we use only f-strings and nothing else? No, there are certain limitations too.
- An f-string is converted into a regular string when it appears in the program.
  - It will never be evaluated again,
  - The expressions (variables) are evaluated only once,
  - If you change the variables embedded in an f-string, the string keeps the original value.

The f-strings

- What’s an f-string? Example:
  - `f'xyz'`,
  - `f"abc"`,
  - `F'foo'`
- An f-string is just a string in which you can embed an expression. Placeholder.
- The expression is evaluated, converted into string form, and inserted right where the expression is.

{expression}

- There must be a way to identify the expression(s).
- Python uses {curly braces} to mark the expression. In most cases, the expressions are variables.
- Any character not inside { } is treated like a regular string.
- F-string expression cannot include a "\".
- Use {{ … }} to for non-escape curry braces.

Restrictions

- Empty expression {} is not allowed.
- An f-string expression can’t contain a backslash (\) character.

```
f’foo{\n}bar’
```

is wrong, but using

```
n = ‘\n’

f’foo{n}bar’
```

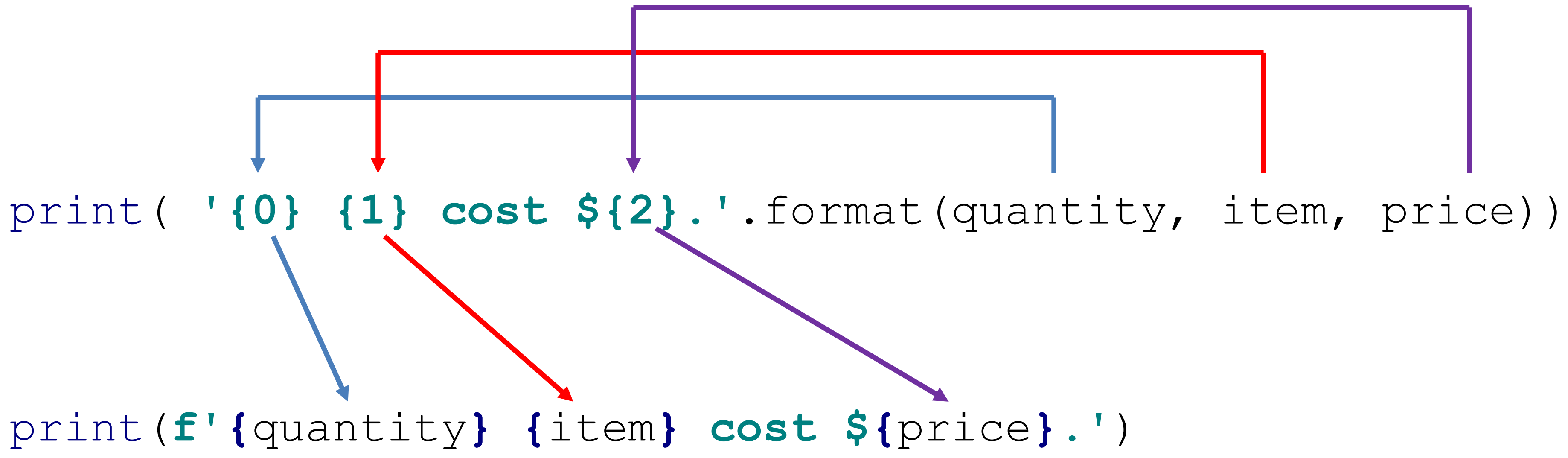
are okay.

Modifiers

- F-strings support extensive modifiers that control the final appearance of the output string.
- The modifier is almost the same as the format() protocol.

A comparison (#2 vs #4)

str.format()



```
print( '{0} {1} cost ${2}.'.format(quantity, item, price))
```

```
print(f'{quantity} {item} cost ${price}.')
```

F-string

Which one is more intuitive?