Strings in Python

→ Strings are ordered sequences of characters

→ We can use indexing & slicing to grab sub-sections of a string

→ Indexing notation uses [] notation after string or variable assigned.

string = "hello"

~~Index~~	h	e	l	l	o
Index	0	1	2	3	4
Rev: Index	-5	-4	-3	-2	-1

For Indexing []

Eg: string[1] & Op:

→ 'h'

Slicing

→ It allows you to grab a subsection of the string.

Syntax:

string_name [start : stop : step]

start: num. index for start

stop: index you will go upto (but not include)

step: size of jump you take

```
s = 'Hello World'

In: s[1:]
O/p: 'ello World'

# Note that there is no change in original s

In: s[:3]
O/p 'Hel'

In: s[:]  # Everything
O/p 'Hello World'

In: s[-1]  # Last letter
O/p 'd'

In: s[:-1]  # Grab everything but last letter
O/p 'Hello Worl'

In: s[::1]  # Grab everything in steps of 1
O/p: 'Hello World'

In: s[::2]  # Grab everything in steps of 2
O/p: 'HloWrd'

In: s[::-1]  # Print string backwards
O/p: 'dlroW olleH'
```

Creating a String

```
Pn: 'a'
o/p: a

Pn: "This is also a string"
o/p: ——— " ——— " ———

Pn: "a"
o/p: 'a'

Pn: 'I'm using single quotes'
o/p: Syntax error

Pn: "I'm using double quotes"
o/p ——— " ——— " ———
```

Printing a String

```
Pn: "Hello"
o/p: 'Hello'

Pn: 'Helo 1'
    'Helo 2'   # We can't o/p multiple strings
                 this way
o/p: 'Helo 2'

Pn: print('a')
    print('b')
o/p: a
     b
```

String Properties

Strings are immutable. i.e. once it's created elements within it cannot be changed

```
In  : s = 'Hello World'
out : Hello World
```

```
In  : s[0] = 'x'
out : TypeError: 'str' object doesn't support item assignment
```

CONCATENATION

```
In  : s + 'concatenate me!'
o/p : 'Hello World concatenate me!'
```

REASSIGNMENT

```
In  : s = s + 'concatenate me!'
fp  : print(s)

o/p : Hello World concatenate me!
```

REPITITION

```
In  : let = 'z'
      let * 3
o/p : 'zzz'
```

Built-in Methods

Methods are in the form:

object.method(parameters)

qn: s = 'Hello world' concatenate me!'

UPPER CASE

qn: s.upper()
o/p: 'HELLO WORLD CONCATENATE ME!'

LOWER CASE

qn: s.lower()
o/p: 'hello world concatenate me!'

SPLIT A STRING BY BLANK SPACES

qn: s.split()
o/p: ['Hello', 'world', 'concatenate', 'me!']

SPLIT BY ELEMENT (Does not include the element split on)

qn: s.split('W')
o/p: ['Hello', 'orld concatenate me!']

DECIMAL FORMATTING

Syntax:

{value:width.precision2.f}

For formatting pi(3.1415926) to 2 decimal places - we'll set the width to 1 because we don't need padding, & precision to 3, giving us the one rq. to the left of decimal & 2 rq.s to the right

```
>>> print(f"Pi to two decimal places is {3.1415926:1.3f}")
Pi to two decimal places is 3.14
```

We'll break it into variables to make it more clear

```
>>> value = 3.1415926
>>> width = 1
>>> precision = 3
>>> print(f"Pi to 2 decimal places is:
                {value:{width}.{precision}f}")
Pi to 2 decimal places is 3.14
```

Let's change the width to 10

```
>>> value = 3.1415926
>>> width = 10
>>> precision = 3
>>> print(f"Pi to 2 decimal places is:
                {value:{width}.{precision}f}")
```

Pi to two decimal places is: 3.14

MULTI LINE STRINGS

Just prepend every line with f

E.g.

```
>>> name = 'Nena'
>>> pi = 3.14
>>> food = 'pee'
>>> message = (
        f"Hello, my name is {name}. "
        f"I can calculate pi to two places:
                                {pi:4.3}. "
f"But I would rather be eating {food}."

    )
>>> print(message)
Hello, my name is Nena. I can calculate
pi to two places: 3.14. But I would rather
be eating pee.
```

TRIMMING A STRING

strip() → removes leading & trailing whitespaces
rstrip() → removes trailing whitespaces
lstrip() → removes leading whitespaces

```
>>> my_string = "   Hello World!   "
>>> print(f">{my_string.lstrip()}<")
>Hello World!   <

>>> print(f">{my_string.rstrip()}<")
>   Hello World!<

>>> print(f">{my_string.strip()}<")
>Hello World!<
```

These functions also accept an optional argument of characters to remove. Let's remove all leading & trailing commas

```
>>> string = "Hello, World!,,,  "
>>> print(string.strip(",", " "))
Hello World!
```

REPLACING CHARACTERS

~~replace("orig-string~~

replace("word_of_string", "replacement")

For e.g.

```
string = "Hello, world!"
string.replace("world", "Nina")
'Hello, Nina!
```

PRINT FORMATTING

~~form~~
format() method to add formatted objects to printed string statements.

In: 'Insert another string with curly brackets: {}'.format('The inserted string')

o/p: 'Insert another string with curly brackets: The inserted string

It helps you inject items into a string rather than trying to chain items together using commas or string concatenation.

```
player = 'Thomas'
points = 33

'Last night, '+player+' scored '+str(points)+' points.'
                                  # concatenation

f'Last night, {player} scored {points} points.'
formatting                              # string
```

3 ways to perform:

- The oldest method involves placeholders using the modulo % character.
- An improved technique uses the .format() string method.
- The newest method, introduced with Python 3.6, uses formatted string literals, called f-strings.

There are multiple ways to format strings for printing variables in them, this is called as string interpolation

Syntax:

'string here {} then also {}'.format('something1', 'something2')

↑ place of insertion

Formatting with Placeholders

You can use %s to inject strings into your print statements. The modulo % is referred as a string formatting operator.

qn: print("I'm going to inject %s here."%('something'))

o/p: I'm going to inject something here

You can inject multiple items by placing them inside a tuple after % operator

qn: print("I'm going to insert %s text here, & %s text here."%('some', 'more'))

o/p: I'm going to insert some text here, & more text here

You can also pass variable names

qn: x, y = 'some', 'more'

print("I'm going to inject %s here, & %s here."%(x,y))

o/p: I'm going to inject some here, & more here

FORMAT CONVERSION

%s & %r convert any python object to a string using two seperate methods: str() & repr().

Note: %r & repr() delivers string representation of the object, including quotation marks & any escape characters

Qn: ~~print('He said his name was %s.' %'Fred')~~

Qn: print('He said his name was %s.' %'Fred')
" " %r "

o/p: He said his name was Fred
He said his name was 'Fred'.

\t inserts a tale into string

Qn: print('I once caught a fish %s.' %'this \t big')
" " " %r "

o/p: I once caught a fish this big.
" " 'this \t big'.

%s operator converts whatever it sees into a string, including integers & floats.

%d converts n.o.s to integers first, without rounding

Qn: print('I wrote %s programs today.' %3.75)
" " %d "

o/p: I wrote 3.75 programs today.
" 3 "

PADDING & PRECISION OF FLOATS

Floating pt. numbers use the format %.5.2f. Here 5 would be the minimum number of characters the string should contain; these may be padded with whitespace if entered no. does not have this many digits. .2f stands for how many no.s to show past decimal pt.

```
Gn: print('Floating pt. nos: %.5.2f' %.(13.144))
```

```
o/p: Floating pt. nos: 13.14
```

MULTIPLE FORMATTING

```
Gn: print('First: %s, Second: %.5.2f, Third: %r'
          %('hi!', 3.1415, 'bye!'))
```

```
o/p: First: hi!, Second: 3.14, Third: 'bye!'
```

.format() method

Syntax:

```
'string here {} then also {}'.format{
('something1', 'something2')
```

Date
Page

Gn: print('This is a string with {}'.format('inserit'))

O/p: This is a string with inserit

(1) Inserted Objects can be called by Index

Gn: print('The {2} {1} {0}'.format('fox', 'brown', 'quick'))

O/p: The quick brown fox

(2) Inserted Objects can be assigned keywords.

Gn: print('First object: {a}, second Object: {b}, Third Object: {c}'.format(a=1, b='Two', c=12.3)

O/p: First Object: 1, second Object: Two, Third Object: 12.3

(3) Inserted objects can be reused

Gn: print('A %s saved is a %s earned.' %('penny', 'penny'))

~~print('A %s saved is a %s~~
print('A {p} saved is a {p} earned.'.format(p='penny'))

o/p: A penny saved is a penny earned

A penny saved is a penny earned

ALIGNMENT, PADDING & PRECISION

With the curly braces you can assign field lengths, left/right alignment, rounding parameters & more.

```
print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))
print('{0:8} | {1:9}'.format('Apples', 3.))
print('{0:8} | {1:9}'.format('Oranges', 10))
```

o/p:

Fruit	Quantity
Apples	3.0
Oranges	10

By default, .format() aligns text to the left, numbers to right. You can pass an optional <, ^, or > to set a left, center or right alignment.

```
print('{0:<8} | {1:^8} | {2:>8}'.
      format('Left', 'Center', 'Right'))
print('{0:<8} | {1:^8} | {2:>8}'.format
      (11, 22, 33))
```

Left	Center	Right
11	22	33

You can precede the alignment operator with a padding character.

Qn: print('{0:=<8} | {1:-^8} | {2:.>8}'.format('Left', 'Center', 'Right'))

O/P: Left==== |-Center-| ... Right
11====== |---22---| 33

Field widths & float precision are handled in a way similar to placeholders. The following two print statements are equivalent.

Qn: print('This is my ten-character, two-decimal number: %10.2f' % 13.579)

print('This is my ten-character, two-decimal number: {0:10.2f}'.format(13.579))

O/p: This is my ten-character, two-decimal no: 13.58
——11—— 11 —— 11 —— 11 —— 11——

Note & 5 of 10 char

Note that there are 5 spaces following the colon, and 5 characters taken up by 13.58, for a total of 10 characters.

FORMATTED STRING LITERALS

(f-strings)

Eg: name = 'Fred'

print(f"He said his name is {name}.")

O/P: He said his name is Fred.

Pass !r to get the string representation

Eg: print(f"He said his name is {name!r}")

O/P: He said his name is 'Fred'

FLOAT FORMATTING

~~Result~~

Syntax:

"result: {value: {width}.{precision}}"

Eg: num = 23.45678

print("My 10 character, four decimal eg. is: {0:10.4f}".format(num))

print(f"My 10 character, four decimal numbers is: {num:{10}.{6}}")

o/p: My 10 character, four decimal ng. is:
____23.4568
-11 ——— 11 ——— 11 ——— 11 ———

Note that with f-strings, precision refers to the ~~th~~ total ng. of digits, not just those foll. the decimal. They fit more closely with scientific notation & statistical analysis. Unfortunately f-strings do not pad to the ~~right~~ of decimal, even if precision allows it.

Qn: num = 23.45
print ~~pf~~("My 10 character, four decimal ng.
is :{0:10.4f}", format(num))

print(f"My 10 charac., four decimal ng.
is :{num:{10}.{6}}")

o/p: My 10 character, four decimal ng. is: 23.4500
← 11 ——— 11 ——— : 23.45

Qn: num = 23.45
print("My 10 charact., four decimal ng. is:
{0:10.4f}", format(num))
print("My 10 charac., four decimal ng. is
:{num:10.4f}")

o/p: My 10 charac., four dec. ng. is: 23.4500
——— 11 ——— 11 ——— 11 ———