```
!python --version
```

```
Python 3.8.16
```

Comment with

You mark a comment by using the # character; everything from that point on to the end of the current line is part of the comment.

You'll usually see a comment on a line by itself, as shown here:

```
# 60 sec/min * 60 min/hr * 24 hr/day
seconds_per_day = 86400
```

Or, on the same line as the code it's commenting:

```
seconds_per_day = 86400 # 60 sec/min * 60 min/hr * 24 hr/day
```

The # character has many names: hash, sharp, pound, or the sinister-sounding octo-thorpe.

Whatever you call it,its effect lasts only to the end of the line on which it appears.

Python does not have a multiline comment. You need to explicitly begin each comment line or section with a #:

```
# I can say anything here, even if Python doesn't like it,
# because I'm protected by the awesome
# octothorpe.
```

However, if it's in a text string, the mighty hash reverts back to its role as a plain old # character:

```
print("No comment: quotes make the # harmless.")
```

```
No comment: quotes make the # harmless.
```

Continue Lines with \

Programs are more readable when lines are reasonably short.

The recommended (not required) maximum line length is 80 characters.

If you can't say everything you want to say in that length, you can use the continuation character: \ (backslash).

Just put \ at the end of a line, and Python will suddenly act as though you're still on the same line.

For example, if I wanted to add the first five digits, I could do it a line at a time:

```
sum = 0
sum += 1
sum += 2
sum += 3
sum += 4
sum
```

```
10
```

Or, I could do it in one step, using the continuation character:

```
sum = 1 + \
      2 + \
      3 + \
      4

sum
```

```
10
```

Compare with if, elif, and else

Our first example is this tiny Python program that checks the value of the boolean variable disaster and prints an appropriate comment:

```
disaster = True
if disaster:
  print("Woe!")
else:
  print("Whee!")
```

```
Woe!
```

The if and else lines are Python statements that check whether a condition (here,the value of disaster) is a boolean True value, or can be evaluated as True.

Remember, print() is Python's built-in function to print things, normally to your screen.

Each print() line is indented under its test.

The recommended style, called PEP-8, is to use four spaces.

```
furry = True
large = True
if furry:
  if large:
    print("It's a yeti.")
  else:
    print("It's a cat!")

else:
  if large:
    print("It's a whale!")
  else:
    print("It's a human. Or a hairless cat.")
```

```
It's a yeti.
```

In Python, indentation determines how the if and else sections are paired.

Our first test was to check furry.

Because furry is True, Python goes to the indented if large test.

Because we had set large to True, if large is evaluated as True, and
the following else line is ignored.

This makes Python run the line indented under if large: and print It's a yeti.

If there are more than two possibilities to test, use if for the first,
elif (meaning else if) for the middle ones, and else for the last:

```
color = "mauve"
if color == "red":
  print("It's a tomato")
elif color == "green":
  print("It's a green pepper")
elif color == "bee purple":
  print("I don't know what it is, but only bees can see it")
else:
  print("I've never heard of the color", color)
```

```
I've never heard of the color mauve
```

In the preceding example, we tested for equality by using the == operator.

Here are Python's comparison operators:

equality	==
inequality	!=
less than	<
less than or equal	<=
greater than	>
greater than or equal	>=

These return the boolean values True or False. Let's see how these all
work, but first, assign a value to x:

```
x = 7
```

```
x == 5
```

```
False
```

```
x == 7
```

```
True
```

```
5 < x
```

```
True
```

```
x < 10
```

```
True
```

Note that two equals signs (==) are used to test equality; remember, a single equals sign (=) is what you use to assign a value to a variable.

If you need to make multiple comparisons at the same time, you use the logical (or boolean) operators and, or, and not to determine the final boolean result.

```
5 < x and x < 10
```

```
True
```

```
(5 < x) and (x < 10)
```

```
True
```

```
5 < x or x < 10
```

```
True
```

```
5 < x and x > 10
```

```
False
```

```
5 < x and not x > 10
```

```
True
```

If you're anding multiple comparisons with one variable, Python lets you do this:

```
5 < x < 10
```

```
True
```

```
5 < x < 10 < 999
```

```
True
```

What Is True?

A false value doesn't necessarily need to explicitly be a boolean False.

For example, these are all considered False:

```
boolean              False
null                      None
zero integer      0
zero float           0.0
empty string     ''
empty list          []
empty tuple      ()
empty dict         {}
empty set          set()
```

Anything else is considered True.

```
some_list = []
if some_list:
```

```
  print("There's something in here")
else:
  print("Hey, it's empty!")
```

```
Hey, it's empty!
```

Do Multiple Comparisons with in

Suppose that you have a letter and want to know whether it's a vowel.

One way would be to write a long if statement:

```
letter = 'o'
if letter == 'a' or letter == 'e' or letter == 'i' \
   or letter == 'o' or letter == 'u':
   print(letter, 'is a vowel')
else:
  print(letter, 'is not a vowel')
```

```
o is a vowel
```

Whenever you need to make a lot of comparisons like that, separated by or, use Python's membership operator in, instead.

Here's how to check vowel-ness more Pythonically, using in with a string made of vowel characters

```
vowels = 'aeiou'
letter = 'o'
letter in vowels
```

```
True
```

```
if letter in vowels:
  print(letter, 'is a vowel')
```

```
o is a vowel
```

Walrus Operator: Newly included in Python 3.8

Arriving in Python 3.8 is the walrus operator, which looks like this:

```
name := expression
```

Normally, an assignment and test take two steps:

```
tweet_limit = 280
tweet_string = "Blah" * 50
diff = tweet_limit - len(tweet_string)
if diff >= 0:
  print("A fitting tweet")
else:
  print("Went over by", abs(diff))
```

```
A fitting tweet
```

With our new walrus operator (aka assignment expressions) we can combine these into one step:

```
tweet_limit = 280
tweet_string = "Blah" * 50
if diff := tweet_limit - len(tweet_string) >= 0:  # value assignment and testing(condition checking) takes one step
  print("A fitting tweet")
else:
  print("Went over by", abs(diff))
```

```
A fitting tweet
```

In Class Exercises:

1 Choose a number between 1 and 10 and assign it to the variable secret. Then, select another number between 1 and 10 and assign it to the variable guess. Next,write the conditional tests (if, else, and elif) to print the string 'too low' if guess is less than secret, 'too high' if greater than secret, and 'just right' if equal to secret.

2 Assign True or False to the variables small and green. Write some if/else statements to print which of these matches those choices: cherry, pea, watermelon,pumpkin.

Python Input: Take Input from User

In Python 3, we have the following two built-in functions to handle input from a user and system.

input(prompt): To accept input from a user.
print(): To display output on the console/screen.

Python Example to Accept Input From a User

Let see how to accept employee information from a user.

First, ask employee name, salary, and company name from the user

Next, we will assign the input provided by the user to the variables

Finally, we will use the print() function to display those variables on the screen.

```
# take three values from user
name = input("Enter Employee Name: ")
salary = input("Enter salary: ")
company = input("Enter Company name: ")

# Display all values on screen
print("\n")
print("Printing Employee Details")
```

```
print("Name", "Salary", "Company")
print(name, salary, company)
```

```
Enter Employee Name: AP
Enter salary: 100
Enter Company name: JIS


Printing Employee Details
Name Salary Company
AP 100 JIS
```

How input() Function Works

```
Syntax:    input([prompt])

The prompt argument is optional. The prompt argument is used to display a message to the user. For example, the prompt is, “Please enter your name

When the input() function executes, the program waits until a user enters some value.

Next, the user enters some value on the screen using a keyboard.

Finally, The input() function reads a value from the screen, converts it into a string, and returns it to the calling program.
```

```
number = input("Enter roll number ")
name = input("Enter age ")

print("\n")
print('Roll number:', number, 'Name:', name)
print("Printing type of a input values")
print("type of number", type(number))
print("type of name", type(name))
```

```
Enter roll number 10
Enter age 20


Roll number: 10 Name: 20
Printing type of a input values
type of number <class 'str'>
type of name <class 'str'>
```

Take an Integer Number as input from User

```
# program to calculate addition of two input integer numbers

# convert inout into int
first_number = int(input("Enter first number "))
second_number = int(input("Enter second number "))

print("\n")
print("First Number:", first_number)
print("Second Number:", second_number)
sum1 = first_number + second_number
print("Addition of two number is: ", sum1)
```

```
Enter first number 10
Enter second number 20


First Number: 10
Second Number: 20
Addition of two number is:  30
```

Take Float Number as a Input from User

```
marks = float(input("Enter marks "))
print("\n")
print("Student marks is: ", marks)
print("type is:", type(marks))
```

```
Enter marks 10.3


Student marks is:  10.3
type is: <class 'float'>
```

Get Multiple inputs From a User in One Line

In Python, It is possible to get multiple values from the user in one line.

We can accept two or three values from the user.

For example, in a single execution of the input() function, we can ask the user his/her name, age, and phone number and store it in three different variables.

Let’ see how to do this.

Take each input separated by space
Split input string using split() get the value of individual input

```
name, age, marks = input("Enter your Name, Age, Percentage separated by space ").split()  #put space separated inputs
print("\n")
print("User Details: ", name, age, marks)
```

```
Enter your Name, Age, Percentage separated by space AP 10 24


User Details:  AP 10 24
```

Accept Multiline input From a User

As you know, the input() function does not allow the user to provide values separated by a new line.

If the user tries to enter multiline input, it reads only the first line. Because whenever the user presses the enter key, the input function reads information provided by the user and stops execution.

We can use a loop. In each iteration of the loop, we can get input strings from the user and join them. You can also concatenate each input string

```
# list to store multi line input
# press enter two times to exit
data = []
print("Tell me about yourself")
while True:
    line = input()
    if line:
        data.append(line)
    else:
        break
finalText = '\n'.join(data)
```

```
print("\n")
print("Final text input")
print(finalText)
```

```
     Tell me about yourself
     kjsdaksdjhaksdjfhaskdfjhskdfjlk
     hksjfdkfjsdkfsfdksdfksdl
     ksdkfskdfsfdksjfdkj
     kjdfhksdjfhskdfhsakd
     sdfskdfskhfksh
     sdjfksdjfhksfh



     Final text input
     kjsdaksdjhaksdjfhaskdfjhskdfjlk
     hksjfdkfjsdkfsfdksdfksdl
     ksdkfskdfsfdksjfdkj
     kjdfhksdjfhskdfhsakd
     sdfskdfskhfksh
     sdjfksdjfhksfh
```

Output in Python

Python has a built-in print() function to display output to the standard output device like screen and console.

```
# take input
name = input("Enter Name: ")

# Display output
print('User Name:', name)
```

```
     Enter Name: ap
     User Name: ap
```

```
name = input('Enter Name ')
zip_code = int(input('Enter zip code '))
street = input('Enter street name ')
house_number = int(input('Enter house number '))

# Display all values separated by hyphen
print(name, zip_code, street, house_number, sep="-")
```

```
     Enter Name ap
     Enter zip code 23
     Enter street name dsds
     Enter house number 43
     ap-23-dsds-43
```

Output Formatting

You can display output in various styles and formats using the following functions.

str.format()
repr()
str.rjust(), str.ljust() , and str.center().
str.zfill()
The % operator can also use for output formatting

str.format() to format output

```
print('FirstName - {0}, LastName - {1}'.format('Ault', 'Kelly'))

# Here {0} and {1} is the numeric index of a positional argument present in the format method. i.e., {0} = Ault and {1} = Kelly.
# Anything that not enclosed in braces {} is considered a plain literal text.
```

```
FirstName - Ault, LastName - Kelly
```

Format Output String by its positions

```
firstName = input("Enter First Name ")
lastName = input("Enter Last Name ")
organization = input("Enter Organization Name ")

print("\n")
print('{0}, {1} works at {2}'.format(firstName, lastName, organization))
print('{1}, {0} works at {2}'.format(firstName, lastName, organization))
print('FirstName {0}, LastName {1} works at {2}'.format(firstName, lastName, organization))
print('{0}, {1} {0}, {1} works at {2}'.format(firstName, lastName, organization))
```

```
Enter First Name Apurba
Enter Last Name Paul
Enter Organization Name JISCE


Apurba, Paul works at JISCE
Paul, Apurba works at JISCE
FirstName Apurba, LastName Paul works at JISCE
Apurba, Paul Apurba, Paul works at JISCE
```

Accessing Output String Arguments by name

```
name = input("Enter Name ")
marks = input("Enter marks ")

print("\n")
print('Student: Name:  {firstName}, Marks: {percentage}%'.format(firstName=name, percentage=marks))
```

```
Enter Name Apurba
Enter marks 100


Student: Name:  Apurba, Marks: 100%
```

Output Alignment by Specifying a Width

```
text = input("Enter text ")

print("\n")
# left aligned
print('{:<25}'.format(text))
# Right aligned
print('{:>25}'.format(text))
# centered
print('{:^25}'.format(text))
```

```
Enter text This is a sample text


This is a sample text
    This is a sample text
  This is a sample text
```

Specifying a Sign While Displaying Output Numbers

```
positive_number = float(input("Enter Positive Number "))
negative_number = float(input("Enter Negative Number "))
```

```
print("\n")
# sign '+' is for both positive and negative number
print('{:+f}; {:+f}'.format(positive_number, negative_number))

# sign '-' is only for negative number
print('{:f}; {:-f}'.format(positive_number, negative_number))
```

```
Enter Positive Number 10.8
Enter Negative Number -9.08


+10.800000; -9.080000
10.800000; -9.080000
```

Display Output Number in Various Format

```
number = int(input("Enter number "))

print("\n")
# 'd' is for integer number formatting
print("The number is:{:d}".format(number))

# 'o' is for octal number formatting, binary and hexadecimal format
print('Output number in octal format : {0:o}'.format(number))

# 'b' is for binary number formatting
print('Output number in binary format: {0:b}'.format(number))

# 'x' is for hexadecimal format
print('Output number in hexadecimal format: {0:x}'.format(number))

# 'X' is for hexadecimal format
print('Output number in HEXADECIMAL: {0:X}'.format(number))
```

```
Enter number 123


The number is:123
Output number in octal format : 173
Output number in binary format: 1111011
Output number in hexadecimal format: 7b
Output number in HEXADECIMAL: 7B
```

Display Numbers as a float type

```
number = float(input("Enter float Number "))

print("\n")
# 'f' is for float number arguments
print("Output Number in The float type :{:f}".format(number))

# padding for float numbers
print('padding for output float number{:5.2f}'.format(number))

# 'e' is for Exponent notation
print('Output Exponent notation{:e}'.format(number))

# 'E' is for Exponent notation in UPPER CASE
print('Output Exponent notation{:E}'.format(number))
```

```
Enter float Number 10.56


Output Number in The float type :10.560000
padding for output float number10.56
Output Exponent notation1.056000e+01
Output Exponent notation1.056000E+01
```

Output String Alignment

```
text = input("Enter String ")

print("\n")
print("Left justification", text.ljust(60, "*"))
print("Right justification", text.rjust(60, "*"))
print("Center justification", text.center(60, "*"))
```

```
Enter String JIS COLLEGE OF ENGINEERING


Left justification JIS COLLEGE OF ENGINEERING**********************************
Right justification **********************************JIS COLLEGE OF ENGINEERING
Center justification *****************JIS COLLEGE OF ENGINEERING*****************
```

Python has three ways of formatting strings:

- old style (supported in Python 2 and 3)
- new style (Python 2.6 and up)
- f-strings (Python 3.6 and up)

Old style: %

The old style of string formatting has the form format_string % data. Inside the format string are interpolation sequences.

Table 5-1 illustrates that the very simplest sequence is a % followed
by a letter indicating the data type to be formatted.

Table 5-1. Conversion types

%s string
%d decimal integer
%x hex integer
%o octal integer
%f decimal float
%e exponential float
%g decimal or exponential float
%% a literal %

You can use a %s for any data type, and Python will format it as a string with no extra spaces.

Following are some simple examples. First, an integer:

```
x='%s' % 42
print(x)
print(type(x))
```

```
42
<class 'str'>
```

```
x='%d' % 42
print(x)
print(type(x))
```

```
42
<class 'str'>
```

```
'%x' % 42
```

```
'2a'
```

```
'%o' % 42
```

```
'52'
```

Next A float:

```
'%s' % 7.03
```

```
'7.03'
```

```
'%f' % 7.03
```

```
'7.030000'
```

```
'%e' % 7.03
```

```
'7.030000e+00'
```

```
'%g' % 7.03
```

```
'7.03'
```

An integer and a literal %:

```
'%d%%' % 100
```

```
'100%'
```

Let’s try some string

```
actor = 'Richard Gere'
cat = 'Chester'
weight = 28
"My wife's favorite actor is %s" % actor
```

```
'My wife's favorite actor is Richard Gere'
```

```
"Our cat %s weighs %s pounds" % (cat, weight)
```

```
'Our cat Chester weighs 28 pounds'
```

That %s inside the string means to interpolate a string.

The number of % appearances in the string needs to match the number of data items after the % that follows the string.

A single data item such as actor goes right after that final %.

Multiple data must be grouped into a tuple such as (cat, weight).

You can add other values in the format string between the % and the type specifier to designate minimum and maximum widths, alignment, and character filling.

Let’s take a quick look at these values:

- An initial '%' character.

- An optional alignment character: nothing or '+' means right-align, and '-' means left-align.
- An optional minwidth field width to use.
- An optional '.' character to separate minwidth and maxchars.
- An optional maxchars (if conversion type is s) saying how many characters to print from the data value. If the conversion type is f, this specifies precision (how many digits to print after the decimal point).
- The conversion type character from the earlier table.

```
thing = 'woodchuck'
```

```
'%s' % thing
```

```
'woodchuck'
```

```
'%12s' % thing
```

```
'   woodchuck'
```

```
'%+12s' % thing
```

```
'   woodchuck'
```

```
'%-12s' % thing
```

```
'woodchuck   '
```

```
'%.3s' % thing   #An optional '.' character to separate minwidth and maxchars.
```

```
'woo'
```

```
'%12.3s' % thing  #An optional '.' character to separate minwidth and maxchars.
```

```
'         woo'
```

```
'%-12.3s' % thing
```

```
'woo         '
```

Once more with feeling, and a float with %f variants:

```
thing = 98.6
```

```
'%f' % thing
```

```
'98.600000'
```

```
'%12f' % thing
```

```
'   98.600000'
```

```
'%+12f' % thing
```

```
'  +98.600000'
```

```
'%-12f' % thing
```

```
'98.600000   '
```

```
'%.3f' % thing
```

```
'98.600'
```

```
'%12.3f' % thing
```

```
'      98.600'
```

```
'%-12.3f' % thing
```

```
'98.600      '
```

And an integer with %d:

```
thing = 9876
```

```
'%d' % thing
```

```
'9876'
```

```
'%12d' % thing
```

```
'        9876'
```

```
'%+12d' % thing
```

```
'       +9876'
```

```
'%-12d' % thing
```

```
'9876        '
```

```
'%.3d' % thing
```

```
'9876'
```

```
'%12.3d' % thing
```

```
'        9876'
```

```
'%-12.3d' % thing
```

```
'9876        '
```

For an integer, the %+12d just forces the sign to be printed, and the format strings with .3 in them have no effect as they do for a float.

New style: {} and format()

"New style" formatting has the form

format_string.format(data).

```
thing = 'woodchuck'
'{}'.format(thing)
```

```
'woodchuck'
```

The arguments to the format() function need to be in the order as the {} placeholders in the format string:

```
thing = 'woodchuck'
place = 'lake'
'The {} is in the {}.'.format(thing, place)
```

```
'The woodchuck is in the lake.'
```

With new-style formatting, you can also specify the arguments by position like this:

```
'The {1} is in the {0}.'.format(place, thing)  #The value 0 referred to the first argument, place, and 1 referred to thing.
```

```
'The woodchuck is in the lake.'
```

The arguments to format() can also be named arguments

```
'The {thing} is in the {place}'.format(thing='duck', place='bathtub')
```

```
'The duck is in the bathtub'
```

```
#In the following example, {0} is the first argument to format() (the dictionary d):

d = {'thing': 'duck', 'place': 'bathtub'}
'The {0[thing]} is in the {0[place]}.'.format(d)
```

```
'The duck is in the bathtub.'
```

New-style formatting has a slightly different format string definition from the old-style one (examples follow):

- An initial colon (':').
- An optional fill character (default ' ') to pad the value string if it's shorter than minwidth.
- An optional alignment character. This time, left alignment is the default. '<' also means left, '>' means right, and '^' means center.
- An optional sign for numbers. Nothing means only prepend a minus sign ('-') for negative numbers. ' ' means prepend a minus sign for negative numbers, and a space (' ') for positive ones.
- An optional minwidth. An optional period ('.') to separate minwidth and max-chars.
- An optional maxchars.
- The conversion type.

```
thing = 'wraith'
place = 'window'
```

```
'The {} is at the {}'.format(thing, place)
```

```
'The wraith is at the window'
```

```
'The {:10s} is at the {:10s}'.format(thing, place)
```

```
'The wraith     is at the window    '
```

```
'The {:<10s} is at the {:<10s}'.format(thing, place)
```

```
'The wraith     is at the window    '
```

```
'The {:>10s} is at the {:>10s}'.format(thing, place)
```

```
'The     wraith is at the     window'
```

```
'The {:^10s} is at the {:^10s}'.format(thing, place)
```

```
'The   wraith   is at the   window  '
```

Newest Style: f-strings

f-strings appeared in Python 3.6, and are now the recommended way of formatting strings.
To make an f-string:

- Type the letter f or F directly before the initial quote.
- Include variable names or expressions within curly brackets ({}) to get their values into the string.

It's like the previous section's "new-style" formatting, but without the format() function, and without empty brackets ({}) or positional ones ({1}) in the format string.

```
thing = 'wereduck'
place = 'werepond'
f'The {thing} is in the {place}'
```

```
'The wereduck is in the werepond'
```

Expressions are also allowed inside the curly brackets:

```
f'The {thing.capitalize()} is in the {place.rjust(20)}'
```

```
'The Wereduck is in the             werepond'
```

f-strings use the same formatting language (width, padding, alignment) as new-style formatting, after a ':'.

```
f'The {thing:>20} is in the {place:.^20}'
```

```
'The              wereduck is in the ......werepond......'
```

Starting in Python 3.8, f-strings gain a new shortcut that's helpful when you want to print variable names as well as their values.

This is handy when debugging.

The trick is to have a single = after the name in the {}-enclosed part of the f-string:

```
f'{thing =}, {place =}'
```

```
'thing ='wereduck', place ='werepond''
```

Get a Substring with a Slice

You can extract a substring (a part of a string) from a string by using a slice.

You define a slice by using square brackets, a start offset, an end

offset, and an optional step count between them.

You can omit some of these.

The slice will include characters from offset start to one before end:

- [:] extracts the entire sequence from start to end.
- [start :] specifies from the start offset to the end.
- [: end] specifies from the beginning to the end offset minus 1.
- [start : end] indicates from the start offset to the end offset minus 1.
- [start : end : step] extracts from the start offset to the end offset minus 1, skipping characters by step.

Offsets go 0, 1, and so on from the start to the right, and -1,-2, and so forth from the end to the left.

If you don't specify start, the slice uses 0 (the beginning).

If you don't specify end, it uses the end of the string.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
letters[:]
```

```
'abcdefghijklmnopqrstuvwxyz'
```

```
letters[20:]
```

```
'uvwxyz'
```

```
letters[10:]
```

```
'klmnopqrstuvwxyz'
```

```
letters[12:15] #offset 12 through 14
#The start offset is inclusive, and the end offset is exclusive:
```

```
'mno'
```

```
letters[-3:]  #three last characters:
```

```
'xyz'
```

In this next example, we go from offset 18 to the fourth before the end; notice the difference from the previous example, in which starting at -3 gets the x, but ending at -3 actually stops at -4, the w:

```
letters[18:-3]
```

```
'stuvw'
```

```
letters[-6:-2]
```

```
'uvwx'
```

If you want a step size other than 1, specify it after a second colon, as shown in the next series of examples.

From the start to the end, in steps of 7 characters:

```
letters[::7]
```

```
'ahov'
```

```
letters[4:20:3] #From offset 4 to 19, by 3:
```

```
'ehknqt'
```

```
letters[19::4]#From offset 19 to the end, by 4:
```

```
'tx'
```

```
letters[:21:5]#From the start to offset 20 by 5:
```

```
'afkpu'
```

And that’s not all! Given a negative step size, this handy Python slicer can also step backward.

This starts at the end and ends at the start, skipping nothing:

```
letters[-1::-1]
```

```
'zyxwvutsrqponmlkjihgfedcba'
```

It turns out that you can get the same result by using this:

```
letters[::-1]
```

```
'zyxwvutsrqponmlkjihgfedcba'
```

Get Length with len()

The len() function counts characters in a string:

```
len(letters)
```

```
26
```

```
empty = ""
len(empty)
```

```
0
```

Split with split()

```
tasks = 'get gloves,get mask,give cat vitamins,call ambulance'
tasks.split(',')
```

```
['get gloves', 'get mask', 'give cat vitamins', 'call ambulance']
```

```
tasks.split()
#If you don’t specify a separator, split() uses any sequence of white space characters—newlines, spaces, and tabs:
```

```
['get', 'gloves,get', 'mask,give', 'cat', 'vitamins,call', 'ambulance']
```

Combine by Using join()

The join() function is the opposite of split(): it collapses a list of strings into a single string.

It looks a bit backward because you specify the string that glues everything together first, and then the list of strings to glue: string .join(list).

So, to join the list lines with separating newlines, you would say '\n'.join(lines). In the following example, let's join some names in a list with a comma and a space:

```
crypto_list = ['Yeti', 'Bigfoot', 'Loch Ness Monster']
crypto_string = ', '.join(crypto_list)
print('Found and signing book deals:', crypto_string)
```

```
Found and signing book deals: Yeti, Bigfoot, Loch Ness Monster
```

Substitute by Using replace()

You use replace() for simple substring substitution.

Give it the old substring, the new one, and how many instances of the old substring to replace.

It returns the changed string but does not modify the original string.

If you omit this final count argument, it replaces all instances.

In this example, only one string ('duck') is matched and replaced in the returned string:

```
setup = "a duck goes into a bar..."
setup.replace('duck', 'marmoset')
```

```
'a marmoset goes into a bar...'
```

```
setup
```

```
'a duck goes into a bar...'
```

Strip with strip()

It's very common to strip leading or trailing "padding" characters from a string, especially spaces.

The strip() functions shown here assume that you want to get rid of whitespace characters (' ', '\t', '\n') if you don't give them an argument.

strip() strips both ends, lstrip() only from the left, and rstrip() only from the right.

Let's say the string variable world contains the string "earth" floating in spaces:

```
world = "    earth   "
```

```
world.strip()
```

```
'earth'
```

```
world.strip(' ')
```

```
'earth'
```

```
world.lstrip()
```

```
'earth   '
```

```
world.rstrip()
```

```
'    earth'
```

If the character were not there, nothing happens:

```
world.strip('!')
```

```
'    earth   '
```

```
import string
string.whitespace
```

```
' \t\n\r\x0b\x0c'
```

```
string.punctuation
```

```
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

```
blurt = "What the...!!?"
blurt.strip(string.punctuation)
```

```
'What the'
```

```
prospector = "What in tarnation ...??!!"
prospector.strip(string.whitespace + string.punctuation)
```

```
'What in tarnation'
```

Search and Select

```
poem = '''All that doth flow we cannot liquid name
Or else would fire and water be the same;
But that is liquid which is moist and wet
Fire that property can never get.
Then 'tis not cold that doth the fire put out
But 'tis the wet that makes it die, no doubt.'''
```

```
poem[:13]
```

```
'All that doth'
```

```
len(poem)
```

```
250
```

```
poem.startswith('All')
```

```
True
```

```
poem.endswith('That\'s all, folks!')
```

```
False
```

Python has two methods (find() and index()) for finding the offset of a substring, and has two versions of each (starting from the beginning or the end).

They work the same if the substring is found. If it isn't, find() returns -1, and index() raises an exception.

Let's find the offset of the first occurrence of the word the in the poem:

```
word = 'the'
poem.find(word)
```

```
73
```

```
poem.index(word)
```

```
73
```

```
#And the offset of the last the:
poem.rfind(word)
```

```
214
```

```
poem.rindex(word)
```

```
214
```

```
word = "duck"
poem.find(word)  #But if the substring isn't in there
```

```
-1
```

```
poem.rfind(word)
```

```
-1
```

```
poem.index(word)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-147-c4456b6bd3bf> in <module>
----> 1 poem.index(word)

ValueError: substring not found
```

SEARCH STACK OVERFLOW

How many times does the three-letter sequence the occur?

```
word = 'the'
poem.count(word)
```

```
3
```

Are all of the characters in the poem either letters or numbers?

```
poem.isalnum()
```

```
False
```

Case

```
setup = 'a duck goes into a bar...'
```

```
setup.strip('.')
```

```
'a duck goes into a bar'
```

```
setup.capitalize() #Capitalize the first word:
```

```
'A duck goes into a bar...'
```

```
setup.title() #Capitalize all the words:
```

```
'A Duck Goes Into A Bar...'
```

```
setup.upper() #Convert all characters to uppercase:
```

```
'A DUCK GOES INTO A BAR...'
```

```
setup.lower() #Convert all characters to lowercase:
```

```
'a duck goes into a bar...'
```

```
setup.swapcase() #Swap uppercase and lowercase:
```

```
'A DUCK GOES INTO A BAR...'
```

Alignment

```
setup.center(30)#Center the string within 30 spaces:
```

```
'  a duck goes into a bar...   '
```

```
setup.ljust(30) #Left justify:
```

```
'a duck goes into a bar...     '
```

```
setup.rjust(30)#Right justify:
```

```
'     a duck goes into a bar...'
```

In Class Assignment

```
Q1.  Capitalize the word starting with m:
song = """When an eel grabs your arm,
And it causes great harm,
That's - a moray!"""

Q2.Write the following poem by using old-style formatting. Substitute the strings 'roast beef', 'ham', 'head', and 'clam' into this string:
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.

Q3. Write a form letter by using new-style formatting. Save the following string as letter (you'll use it in the next exercise):

Dear {salutation} {name},
Thank you for your letter. We are sorry that our {product}
{verbed} in your {room}. Please note that it should never
be used in a {room}, especially near any {animals}.
Send us your receipt and {amount} for shipping and handling.
We will send you another {product} that, in our tests,
is {percent}% less likely to have {verbed}.
Thank you for your support.
Sincerely,
{spokesman}
{job_title}
```

✓ 0s completed at 7:22 AM