16/06/2025, 01:25

3. An Informal Introduction to Python - Python 3.13.5 documentation

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and ...): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

You can use the "Copy" button (it appears in the upper-right corner when hovering over or tapping a code ex-

ample), which strips prompts and omits output, to copy and paste the input lines into your interpreter.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash charac- ter within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by

Python, they may be omitted when typing in examples.

Some examples:

# *this is the first comment*

spam = **1** # *and this is the second comment*

text =

**#** ... *and now* a *third!*

"# This is not a comment because it's inside quotes.'

3.1. Using Python as a Calculator

**"I**

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It shouldn't take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, \* and / can be used to perform arithmetic; parenthe-

ses (()) can be used for grouping. For example:

**>>>** 2 + 2

4

**>>>** 50 5\*6

20

-

**>>>** (50 - **5**\*6) / 4

5.0

**>>>** 8 **/** 5 # *division always returns* a *floating-point number*

1.6

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type

float. We will see more about numeric types later in the tutorial.

Division (/) always returns a float. To do floor division and get an integer result you can use the // operator; to calculate the remainder you can use %:

https://docs.python.org/3/tutorial/introduction.html

1/9

16/06/2025, 01:25

3. An Informal Introduction to Python Python 3.13.5 documentation

**>>> 17** / 3 # classic *division returns* a *float*

5.666666666666667

>>>

**>>>** 17 // 3

5

**>>>** 17 % 3

2

# *floor division discards the fractional part*

# *the % operator returns the remainder of the division*

**>>>** 5 \* 3 **+** 2 # *floored quotient* \* *divisor* **+** *remainder*

17

With Python, it is possible to use the \*\* operator to calculate powers [1]:

**>>>** 5

25

\*\*

2

# 5 *squared*

# 2 to *the power* of 7

**>>>** 2 \*\* 7 128

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next in- teractive prompt:

**>>>** width

**=** 20

**>>>** height = 5\***9**

**>>>** width \* height

900

If a variable is not "defined" (assigned a value), trying to use it will give you an error:

**>>>** n # *try to access an undefined variable* Traceback (most recent call last):

File "<stdin>", line 1**,** in <module> NameError: name 'n' is not defined

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

\*

**>>>** 4 3.75 1

14.0

In interactive mode, the last printed expression is assigned to the variable. This means that when you are us- ing Python as a desk calculator, it is somewhat easier to continue calculations, for example:

**>>>** tax = **>>>** price

12.5/100

= 100.50

**>>>** price \* tax

12.5625

**>>>** price + 113.0625

**>>>** round(\_, 2)

113.06

This variable should be treated as read-only by the user. Don't explicitly assign a value to it - you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction.

Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part

(e.g. 3+5j).

https://docs.python.org/3/tutorial/introduction.html

2/9

16/06/2025, 01:25

3.1.2. Text

3. An Informal Introduction to Python - Python 3.13.5 documentation

Python can manipulate text (represented by type str, so-called "strings") as well as numbers. This includes characters " !", words "rabbit", names "Paris", sentences "Got your back.", etc. "Yay! :)". They can be en- closed in single quotes (' ...') or double quotes ("...") with the same result [2].

**>>>** 'spam eggs' # *single quotes*

'spam eggs

**>>>** "Paris rabbit got your back :)! Yay!" # *double quotes*

'Paris rabbit got your back :)! Yay!'

**>>> '**1975' # *digits and numerals enclosed* in *quotes are also strings*

'1975'

To quote a quote, we need to "escape" it, by preceding it with \. Alternatively, we can use the other type of quotation marks:

**>>>** 'doesn\'t' # *use* \' *to escape the single quote*... "doesn't"

**>>>** "doesn't" # ...*or* use *double quotes instead*

"doesn't"

**>>>** ""Yes," they said.'

"Yes," they said.'

**>>>** "\"Yes, \" they said."

"Yes," they said. '

**>>> '"**Isn\'t," they said.'

**'"**Isn\'t," they said. '

In the Python shell, the string definition and output string can look different. The print() function produces a

more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

››› S **=**

'First line. \nSecond line.' # *\n means newline*

**›››** S # *without print(*), *special characters are included* in *the string*

'First line. \nSecond line.'

**>>>** print(s) *# with print*(), *special characters are interpreted*, *so* \n *produces new line* First line.

Second line.

If you don't want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by

adding an r before the first quote:

**>>>** print('C:\some\name') C:\some

ame

**#** *here \*n *means newline!*

**>>>** print(r'C:\some\name') **#** *note the* r *before the quote*

C:\some\name

There is one subtle aspect to raw strings: a raw string may not end in an odd number of \ characters; see the FAQ entry for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes:

End-of-line

characters are automatically included in the string, but it's possible to prevent this by adding a \ at the end of

the line. In the following example, the initial newline is not included:

...

or

...

**>>>** print(""**"**\

...

Usage: thingy [OPTIONS]

-h

Display this usage message

https://docs.python.org/3/tutorial/introduction.html

3/9

16/06/2025, 01:25

...

""")

-H hostname

Usage: thingy [OPTIONS]

-h

-H hostname

3. An Informal Introduction to Python Python 3.13.5 documentation

Hostname to connect to

Display this usage message Hostname to connect to

Strings can be concatenated (glued together) with the + operator, and repeated with \*

**>>>** # *3 times* '*un'*, *followed by* '*ium'*

**>>>** 3\* 'un' **+** 'ium'

'unununium'

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically

concatenated.

**>>>** 'Py' 'thon'

'Python'

This feature is particularly useful when you want to break long strings:

==

**>>>** text = ('Put several strings within parentheses

...

**>>>** text

'to have them joined together.')

'Put several strings within parentheses to have them joined together.'

This only works with two literals though, not with variables or expressions:

**>>>** prefix

=

'Py'

**>>>** prefix 'thon' # *can't concatenate* a *variable and* a *string literal*

File "<stdin>", line **1**

prefix 'thon'

^^^^^^

SyntaxError: invalid syntax **>>>** ('un' \* 3) 'ium'

File "<stdin>", line **1**

('un' \* 3) 'ium'

^^^^^

SyntaxError: invalid syntax

If you want to concatenate variables or a variable and a literal, use +:

**>>>** prefix + 'thon'

'Python'

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type;

a character is simply a string of size one:

**>>>** word = 'Python'

**>>>** word[0]

# *character* in position 0

'P'

**>>>** word [5] # *character in position* 5

'n'

Indices may also be negative numbers, to start counting from the right:

https://docs.python.org/3/tutorial/introduction.html

4/9

16/06/2025, 01:25

**>>>** word [-1] # *Last character*

'n'

3. An Informal Introduction to Python Python 3.13.5 documentation

**>>>** word[-2] # *second- Last character*

**'**0'

**>>>** word [-6]

'P'

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing*, slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

**>>>** word[0:2] # *characters* from *position* (*included*) *to* 2 *(excluded)* 'Py'

**>>>** word[2:5] # *characters* from *position* 2 *(included) to* 5 *(excluded)* 'tho'

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to

the size of the string being sliced.

**>>>** word[:2] 'Py'

>>> word [4:] 'on'

# *character* from *the beginning to position* 2 *(excluded)*

# *characters* from *position* 4 *(included*) to *the end*

**>>>** word[-2:] # *characters* from *the second-Last (included) to the end* 'on'

Note how the start is always included, and the end always excluded. This makes sure that s[i**]** + s[i:] is al-

ways equal to s:

**>>>** word[:2] **+** word [2:] 'Python'

**>>>** word[:4] + word [4:] 'Python'

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left

edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n,* for example:

**-+**

Python |

**+**---

----**+**

0 1 2 3 4 5 6

-6

-5 -4 -3 -2 **-1**

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corre- sponding negative indices. The slice from i to *j* consists of all characters between the edges labeled *i* and *j*,

respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For ex-

ample, the length of word[1:3] is 2.

Attempting to use an index that is too large will result in an error:

**>>>** word [42] # *the word only has* 6 *characters* Traceback (most recent call last):

https://docs.python.org/3/tutorial/introduction.html

5/9

16/06/2025, 01:25

3. An Informal Introduction to Python Python 3.13.5 documentation

File "<stdin>", line **1,** in <module> IndexError: string index out of range

However, out of range slice indexes are handled gracefully when used for slicing:

**>>>** word [4:42]

**'**on'

**>>>** word [42:]

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the

string results in an error:

**>>>** word[0]

= 'J**'**

Traceback (most recent call last):

File "<stdin>", line **1,** in **<**module>

TypeError: 'str' object does not support item assignment

**>>>** word[2:] = 'py'

Traceback (most recent call last):

File "<stdin>", line **1,** in <module>

TypeError: 'str' object does not support item assignment

If you need a different string, you should create a new one:

**>>>** 'J' + word[1:]

'Jython'

**>>>** word[:2] + 'py' 'Pypy'

The built-in function len() returns the length of a string:

**>>>** s = 'supercalifragilisticexpialidocious' **>>>** len(s)

34

**See also:**

**Text Sequence Type — str**

Strings are examples of *sequence types,* and support the common operations supported by such types.

**String Methods**

Strings support a large number of methods for basic transformations and searching.

**f-strings**

String literals that have embedded expressions.

**Format String Syntax**

Information about string formatting with str.format().

**printf-style String Formatting**.

The old formatting operations invoked when strings are the left operand of the % operator are de-

scribed in more detail here.

3.1.3. Lists

https://docs.python.org/3/tutorial/introduction.html

6/9

16/06/2025, 01:25

3. An Informal Introduction to Python Python 3.13.5 documentation

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list,* which can be written as a list of comma-separated values (items) between square brackets. Lists might

contain items of different types, but usually the items all have the same type.

**>>>** squares **›››** squares

=

[1**,** 4, 9, 16, 25]

**[1, 4,** 9, 16, 25]

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

**>>>** squares [0] # *indexing returns the item*

1

**>>>** squares [-1]

25

**>>>** squares [-3:] # *slicing returns* a *new list*

[9, 16, 25]

Lists also support operations like concatenation:

**›››** squares + [36, 49, 64, 81**,** 100]

**[1,** 4, 9, 16, 25, 36, 49, 64, 81**,** 100]

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

**>>>** cubes **>>>** 4 \*\* 3 64

=

**>>>** cubes [3]

**>>>** cubes

[1**,** 8, 27, 65, 125] # *something's wrong here*

# the *cube* of 4 *is 64*, not *65!*

= 64 # *replace the wrong value*

**[1,** 8, 27, 64, 125]

You can also add new items at the end of the list, by using the list.append() *method* (we will see more about

methods later):

**>>>** cubes.append(216)

# *add the cube of 6*

**>>>** cubes.append(7 \*\* 3) # *and the cube of 7*

**>>>** cubes

**[1,** 8, 27, 64, 125, 216, 343]

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that

refer to it.:

**>>>** rgb **>>>** rgba

=

["Red", "Green", "Blue"]

=

**>>>** id(rgb)

rgb

==

id(rgba) # *they reference the same object*

True

**>>>** rgba.append("Alph")

**>>>** rgb

["Red", "Green", "Blue", "Alph"]

All slice operations return a new list containing the requested elements. This means that the following slice re-

turns a shallow copy of the list:

**>>>** correct\_rgba = rgba[:]

=

**>>>** correct\_rgba [-1] =

"Alpha"

https://docs.python.org/3/tutorial/introduction.html

7/9

16/06/2025, 01:25

**>>>** correct\_rgba

3. An Informal Introduction to Python - Python 3.13.5 documentation

["Red", "Green", "Blue", "Alpha"] **>>>** rgba

["Red", "Green", "Blue", "Alph"]

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

**>>>** letters

=

['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** letters

['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** # *replace some values*

**>>>** letters [2:5] = ['C', 'D', 'E']

**>>>** letters

['a', 'b', 'C', 'D', 'E', 'f', 'g']

**>>> #** *now remove them*

**>>>** letters [2:5] = []

**>>>** letters

['a', 'b', 'f', 'g']

**>>>** # *clear the list by replacing all the elements with an empty list* **>>>** letters[:]

**>>>** letters

[]

=

[ ]

The built-in function len() also applies to lists:

**>>>** letters =

['a', 'b', 'c', 'd']

**>>>** len(letters)

4

It is possible to nest lists (create lists containing other lists), for example:

>>> a

=

['a', 'b', 'c']

>>> n

=

[1**,** 2, 3]

**>>>** X = [a, n]

>>> X

[['a', 'b', 'c'], **[1,** 2, 3]]

**>>>** x[0]

['a', 'b', 'c']

**>>>** x[0][1]

'b'

3.2. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we

can write an initial sub-sequence of the Fibonacci series as follows:

**>>>** # *Fibonacci series:*

**>>>** # *the* sum *of two elements defines the next*

**>>>** a, b = 0, **1**

J

**>>> while** a **<** 10:

...

0

print(a)

a, b = b, a+b

**1**

1

2

**3** 5000

8

https://docs.python.org/3/tutorial/introduction.html

8/9

16/06/2025, 01:25

3. An Informal Introduction to Python Python 3.13.5 documentation

This example introduces several new features.

•

The first line contains a *multiple assignment:* the variables a and b simultaneously get the new values 0 and

1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evalu- ated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

The while loop executes as long as the condition (here: a < 10) remains true. In Python, like in C, any non- zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple

comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than),

==

(equal to), **<**= (less than or equal to), **>**= (greater than or equal to) and != (not equal to).

• The *body* of the loop is *indented:* indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more compli-

cated input for Python with a text editor; all decent text editors have an auto-indent facility. When a com-

pound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be

indented by the same amount.

• The print() function writes the value of the argument(s) it is given. It differs from just writing the expres-

sion you want to write (as we did earlier in the calculator examples) in the way it handles multiple argu- ments, floating-point quantities, and strings. Strings are printed without quotes, and a space is inserted be-

tween items, so you can format things nicely, like this:

**>>> i** = 256\*256

**>>>** print('The value of **i** is', i)

The value of i is 65536

The keyword argument *end* can be used to avoid the newline after the output, or end the output with a dif- ferent string:

**>>>** a, b =

0,

1

>>> **while** a < 1000:

...

print(a, end=',') a, b = b, a+b

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,

**Footnotes**

[1] Since \*\* has higher precedence than -, -3\*\*2 will be interpreted as - (3\*\*2) and thus result in -9. To

avoid this and get 9, you can use (-3)\*\*2.

.. •

[2] Unlike other languages, special characters such as \n have the same meaning with both single (' ')

and double ("...") quotes. The only difference between the two is that within single quotes you don't

need to escape

(but you have to escape \') and vice versa.

https://docs.python.org/3/tutorial/introduction.html

9/9