STRING

String adalah salah satu jenis yang paling populer dengan Python. Kita bisa membuatnya hanya dengan melampirkan karakter dalam tanda kutip. Python memperlakukan tanda petik tunggal sama dengan tanda kutip ganda. Membuat string semudah memberi nilai pada sebuah variabel. Misalnya -

```
var1 = 'Hello World!'

var2 = "Python Programming"
```

Mengakses Nilai dalam String

Python tidak mendukung tipe karakter; Ini diperlakukan sebagai string dengan panjang satu, sehingga juga dianggap sebagai substring.
Untuk mengakses substring, gunakan tanda kurung siku untuk mengiris beserta indeks atau indeks untuk mendapatkan substring Anda. Misalnya -

```
#!/usr/bin/python

var1 = 'Hello World!'

var2 = "Python Programming"
print "var1[0]: ", var1[0]

print "var2[1:5]: ", var2[1:5]
```

Bila kode diatas dieksekusi, maka menghasilkan hasil sebagai berikut -

```
var1[0]:  H

var2[1:5]:  ytho
```

```
#!/usr/bin/python
```

Memperbarui String

Anda dapat "memperbarui" string yang ada dengan (kembali) menugaskan variabel ke string lain. Nilai baru dapat dikaitkan dengan nilai sebelumnya atau ke string yang sama sekali berbeda sama sekali. Misalnya -

```
var1 = 'Hello World!'


print "Updated String :- ", var1[:6] + 'Python'
```

Bila kode diatas dieksekusi, maka menghasilkan hasil sebagai berikut -

```
Updated String :- Hello Python
```

Karakter melarikan diri

Tabel berikut adalah daftar karakter escape atau non-printable yang dapat diwakili dengan notasi backslash.
Karakter pelarian ditafsirkan; dalam satu dikutip serta dua kali mengutip string.

Backslashnotation	Hexadecimalcharacter	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M- \C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Operator Khusus

Asumsikan variabel string memegang 'Halo' dan variabel b berisi 'Python',
lalu –

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r' \n' prints \n and print R' \n'prints \n
%	Format - Performs String formatting	See at next section

Penyandian String Operator

Salah satu fitur Python yang paling keren adalah format string operator %. Operator ini unik untuk string dan membuat paket memiliki fungsi dari keluarga printf C () C. Berikut adalah contoh sederhana -

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

Bila kode diatas dieksekusi, maka menghasilkan hasil sebagai berikut -

```
My name is Zara and weight is 21 kg!
```

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table −

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	' % %' leaves you with a single literal ' %'
(var)	mapping variable (dictionary arguments)
m.n.	m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Triple Quotes

Tiga tanda kutip Python hadir untuk menyelamatkannya dengan membiarkan string memanjang banyak baris, termasuk kata kunci NEWLINEs, TABs, dan karakter khusus lainnya. Sintaks untuk triple quotes terdiri dari tiga tanda kutip tunggal atau ganda berturut-turut.

```
#!/usr/bin/python

para _str = "" "ini adalah string panjang yang terdiri dari

beberapa baris dan karakter yang tidak dapat dicetak seperti
TAB ( \ t) dan mereka akan muncul seperti itu saat ditampilkan.
NEWLINEs dalam string, apakah secara eksplisit diberikan seperti
Ini dalam tanda kurung [ \ n], atau hanya NEWLINE di dalamnya
tugas variabel juga akan muncul.
"" "
Cetak para _str
```

Bila kode diatas dieksekusi, maka hasilnya akan menghasilkan hasil berikut. Perhatikan bagaimana setiap karakter khusus telah diubah menjadi bentuk cetaknya, sampai ke NEWLINE terakhir di akhir string antara "up". Dan menutup tanda kutip tiga kali. Perhatikan juga bahwa NEWLINEs terjadi baik dengan carriage return yang eksplisit di akhir baris atau kode escape-nya (\ n) -

```
Ini adalah string panjang yang terdiri dari
beberapa baris dan karakter yang tidak dapat dicetak seperti
TAB () dan mereka akan muncul seperti itu saat ditampilkan.
NEWLINEs dalam string, apakah secara eksplisit diberikan seperti
ini dalam tanda kurung [
], atau hanya NEWLINE di dalamnya
tugas variabel juga akan muncul.
```

String mentah tidak memperlakukan garis miring terbalik sebagai karakter spesial sama sekali. Setiap karakter yang Anda masukkan ke dalam string mentah tetap seperti yang Anda tulis -

```
#!/usr/bin/python

Cetak 'C: \ \ tempat'
```

Bila kode diatas dieksekusi, maka menghasilkan hasil sebagai berikut -

```
C: di mana-mana
```

Sekarang mari kita gunakan string mentah. Kami akan mengutarakan ekspresi 'sebagai berikut -

```
#! / Usr / bin / python

Cetak r'C: \ \ tempat '
```

Bila kode diatas dieksekusi, maka menghasilkan hasil sebagai berikut -

```
C: tidak di mana-mana
```

String Unicode

String normal dengan Python disimpan secara internal sebagai 8-bit ASCII, sedangkan string Unicode disimpan sebagai Unicode 16-bit. Hal ini memungkinkan untuk serangkaian

karakter yang lebih bervariasi, termasuk karakter khusus dari kebanyakan bahasa di dunia. Saya akan membatasi perlakuan saya terhadap string Unicode sebagai berikut -

#! / Usr / bin / python

Cetak u'Hello, dunia! '
Bila kode diatas dieksekusi, maka menghasilkan hasil sebagai berikut -
Halo Dunia!
Seperti yang Anda lihat, senar Unicode menggunakan awalan Anda, sama seperti senar mentah menggunakan awalan r.

7.1. — Common string operations

Source code: Lib/string.py

The string module contains a number of useful constants and classes, as well as some deprecated legacy functions that are also available as methods on strings. In addition, Python's built-in string classes support the sequence type methods described in the Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange section, and also the string-specific methods described in the String Methods section. To output formatted strings use template strings or the % operator described in the String Formatting Operations section. Also, see the re module for string functions based on regular expressions.

7.1.1. String constants

The constants defined in this module are:
string.ascii _letters
The concatenation of the ascii _lowercase and ascii _uppercase constants described below. This value is not locale-dependent.
string.ascii _lowercase
The lowercase letters 'abcdefghijklmnopqrstuvwxyz'. This value is not locale-dependent and will not change.
string.ascii _uppercase
The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. This value is not locale-dependent and will not change.
string.digits
The string '0123456789'.
string.hexdigits
The string '0123456789abcdefABCDEF'.
string.letters
The concatenation of the strings lowercase and uppercase described below. The specific value is locale-dependent, and will be updated when locale.setlocale() is called.
string.lowercase
A string containing all the characters that are considered lowercase letters. On most systems this is the string 'abcdefghijklmnopqrstuvwxyz'. The specific value is locale-dependent, and will be updated when locale.setlocale() is called.

string.octdigits

The string '01234567'.

string.punctuation

String of ASCII characters which are considered punctuation characters in the C locale.

string.printable

String of characters which are considered printable. This is a combination of digits , letters , punctuation , and whitespace .

string.uppercase

A string containing all the characters that are considered uppercase letters. On most systems this is the string 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. The specific value is locale-dependent, and will be updated when locale.setlocale() is called.

string.whitespace

A string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

7.1.2. Custom String Formatting

New in version 2.6.

The built-in str and unicode classes provide the ability to do complex variable substitutions and value formatting via the str.format() method described in PEP 3101 . The Formatter class in the string module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in format() method.

class string.Formatter

The Formatter class has the following public methods:

format(*format _string*, **args*, ***kwargs*)

The primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. It is just a wrapper that calls vformat() .

vformat(*format _string*, *args*, *kwargs*)

This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the *args and **kwargs syntax. vformat() does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

In addition, the Formatter defines a number of methods that are intended to be replaced by subclasses:

parse(*format _string*)

Loop over the format _string and return an iterable of tuples (*literal _text*, *field _name*, *format _spec*, *conversion*). This is used by vformat() to break the string into either literal text, or replacement fields.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then *literal _text* will be a zero-length string. If there is no replacement field, then the values of *field _name*, *format _spec* and *conversion* will be None.

get _field(*field _name*, *args*, *kwargs*)

Given *field _name* as returned by parse() (see above), convert it to an object to be formatted. Returns a tuple (obj, used _key). The default version takes strings of the form defined in

PEP 3101 , such as ”0[name] ” or ”label.title ”. *args* and *kwargs* are as passed in to vformat() . The return value *used _key* has the same meaning as the *key* parameter to get _value() .

get _value(*key*, *args*, *kwargs*)

Retrieve a given field value. The *key* argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in *args*; if it is a string, then it represents a named argument in *kwargs*.

The *args* parameter is set to the list of positional arguments to vformat() , and the *kwargs* parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; Subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression ‘0.name’ would cause get _value() to be called with a *key* argument of 0. The name attribute will be looked up after get _value() returns by calling the built-in getattr() function.

If the index or keyword refers to an item that does not exist, then an IndexError or KeyError should be raised.

check _unused _args(*used _args*, *args*, *kwargs*)

Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the *args* and *kwargs* that was passed to vformat. The set of unused args can be calculated from these parameters. check _unused _args() is assumed to raise an exception if the check fails.

format _field(*value*, *format _spec*)

format _field() simply calls the global format() built-in. The method is provided so that subclasses can override it.

convert _field(*value*, *conversion*)

Converts the value (returned by get _field()) given a conversion type (as in the tuple returned by the parse() method). The default version understands ‘s’ (str), ‘r’ (repr) and ‘a’ (ascii) conversion types.

7.1.3. Format String Syntax

The str.format() method and the Formatter class share the same syntax for format strings (although in the case of Formatter , subclasses can define their own format string syntax). Format strings contain ”replacement fields ” surrounded by curly braces { }. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: { { and } }.

The grammar for a replacement field is as follows:

replacement _field ::= ” { ” [field _name] [”! ” conversion] [”: ” format _spec] ” } ”
field _name ::= arg _name (”. ” attribute _name | ”[” element _index ”] ”)*
arg _name ::= [identifier | integer]
attribute _name ::= identifier
element _index ::= integer | index _string
index _string ::= <any source character except ”] ”> +
conversion ::= ”r ” | ”s ”
format _spec ::= <described in the next section>

In less formal terms, the replacement field can start with a *field _name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field _name* is optionally followed by a *conversion* field, which is preceded by an exclamation point '!', and a *format _spec*, which is preceded by a colon ':'. These specify a non-default format for the replacement value.
See also the Format Specification Mini-Language section.
The *field _name* itself begins with an *arg _name* that is either a number or a keyword. If it's a number, it refers to a positional argument, and if it's a keyword, it refers to a named keyword argument. If the numerical arg _names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. Because *arg _name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings '10' or ':-]') within a format string. The *arg _name* can be followed by any number of index or attribute expressions. An expression of the form '.name' selects the named attribute using getattr() , while an expression of the form '[index]' does an index lookup using _ _getitem _ _() .
Changed in version 2.7: The positional argument specifiers can be omitted, so ' { } { }' is equivalent to ' {0 } {1 }'.
Some simple format string examples:
"First, thou shalt count to {0 }" # References first positional argument
"Bring me a { }" # Implicitly references the first positional argument
"From { } to { }" # Same as "From {0 } to {1 }"
"My quest is {name }" # References keyword argument 'name'
"Weight in tons {0.weight }" # 'weight' attribute of first positional arg
"Units destroyed: {players[0] }" # First element of keyword argument 'players'.
The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the _ _format _ _() method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling _ _format _ _(), the normal formatting logic is bypassed.
Two conversion flags are currently supported: '!s' which calls str() on the value, and '!r' which calls repr() .
Some examples:
"Harold's a clever {0!s }" # Calls str() on the argument first
"Bring out the holy {name!r }" # Calls repr() on the argument first
The *format _spec* field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own "formatting mini-language " or interpretation of the *format _spec*.
Most built-in types support a common formatting mini-language, which is described in the next section.
A *format _spec* field can also include nested replacement fields within it. These nested replacement fields may contain a field name, conversion flag and format specification, but deeper nesting is not allowed. The replacement fields within the format _spec are substituted before the *format _spec* string is interpreted. This allows the formatting of a value to be dynamically specified.
See the Format examples section for some examples.

7.1.3.1. Format Specification Mini-Language

”Format specifications ” are used within replacement fields contained within a format string to define how individual values are presented (see Format String Syntax). They can also be passed directly to the built-in format() function. Each formattable type may define how the format specification is to be interpreted.
Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.
A general convention is that an empty format string ("") produces the same result as if you had called str() on the value. A non-empty format string typically modifies the result.
The general form of a *standard format specifier* is:
format _spec ::= [[fill]align][sign][#][0][width][,][.precision][type]
fill ::= <any character>
align ::= ”< ” | ”> ” | ”= ” | ” ^ ”
sign ::= ”+ ” | ”- ” | ” ”
width ::= integer
precision ::= integer
type ::= ”b ” | ”c ” | ”d ” | ”e ” | ”E ” | ”f ” | ”F ” | ”g ” | ”G ” | ”n ” | ”o ” | ”s ” | ”x ” | ”X ” | ” % ”
If a valid *align* value is specified, it can be preceded by a *fill* character that can be any character and defaults to a space if omitted. It is not possible to use a literal curly brace (” { ” or ” } ”) as the *fill* character when using the str.format() method. However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn’t affect the format() function.
The meaning of the various alignment options is as follows:

Option	Meaning
’<’	Forces the field to be left-aligned within the available space (this is the default for most objects).
’>’	Forces the field to be right-aligned within the available space (this is the default for numbers).
’=’	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+000000120’. This alignment option is only valid for numeric types. It becomes the default when ‘0’ immediately precedes the field width.
’ ^’	Forces the field to be centered within the available space.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.
The *sign* option is only valid for number types, and can be one of the following:

Option	Meaning
’+’	indicates that a sign should be used for both positive as well as 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.

The ' #' option is only valid for integers, and only for binary, octal, or hexadecimal output. If present, it specifies that the output will be prefixed by '0b', '0o', or '0x', respectively.
The ',' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.
Changed in version 2.7: Added the ',' option (see also PEP 378).
width is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.
When no explicit alignment is given, preceding the *width* field by a zero ('0') character enables sign-aware zero-padding for numeric types. This is equivalent to a *fill* character of '0' with an *alignment* type of '='.
The *precision* is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with 'f' and 'F', or before and after the decimal point for a floating point value formatted with 'g' or 'G'. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer values.
Finally, the *type* determines how the data should be presented.
The available string presentation types are:

Type	Meaning
's'	String format. This is the default type for strings and may be omitted.
None	The same as 's'.

The available integer presentation types are:

Type	Meaning
'b'	Binary format. Outputs the number in base 2.
'c'	Character. Converts the integer to the corresponding unicode character before printing.
'd'	Decimal Integer. Outputs the number in base 10.
'o'	Octal format. Outputs the number in base 8.
'x'	Hex format. Outputs the number in base 16, using lower- case letters for the digits above 9.
'X'	Hex format. Outputs the number in base 16, using upper- case letters for the digits above 9.
'n'	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.
None	The same as 'd'.

In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except 'n' and None). When doing so, float() is used to convert the integer to a floating point number before formatting.
The available presentation types for floating point and decimal values are:

Type	Meaning
’e’	Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent. The default precision is 6.
’E’	Exponent notation. Same as ’e’ except it uses an upper case ‘E’ as the separator character.
’f’	Fixed point. Displays the number as a fixed-point number. The default precision is 6.
’F’	Fixed point. Same as ’f’.
’g’	General format. For a given precision p >= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude.The precise rules are as follows: suppose that the result formatted with presentation type ’e’ and precision p-1 would have exponent exp. Then if -4 <= exp < p, the number is formatted with presentation type ’f’ and precision p-1-exp. Otherwise, the number is formatted with presentation type ’e’ and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it.Positive and negative infinity, positive and negative zero, and nans, are formatted as inf, -inf, 0, -0 and nan respectively, regardless of the precision.A precision of 0 is treated as equivalent to a precision of 1. The default precision is 6.
’G’	General format. Same as ’g’ except switches to ’E’ if the number gets too large. The representations of infinity and NaN are uppercased, too.
’n’	Number. This is the same as ’g’, except that it uses the current locale setting to insert the appropriate number separator characters.
’ %’	Percentage. Multiplies the number by 100 and displays in fixed (’f’) format, followed by a percent sign.
None	The same as ’g’.