STRINGS

UNICODE

Unicode is a system designed to represent *every* character from *every* language. Unicode represents each letter, character, or ideograph as a 4-byte number. Each number represents a unique character used in at least one of the world's languages. (Not all the numbers are used, but more than 65535 of them are, so 2 bytes wouldn't be sufficient.) Characters that are used in multiple languages generally have the same number, unless there is a good etymological reason not to. Regardless, there is exactly 1 number per character, and exactly 1 character per number. Every number always means just one thing; there are no "modes" to keep track of. U+0041 is always 'A', even if your language doesn't have an 'A' in it.

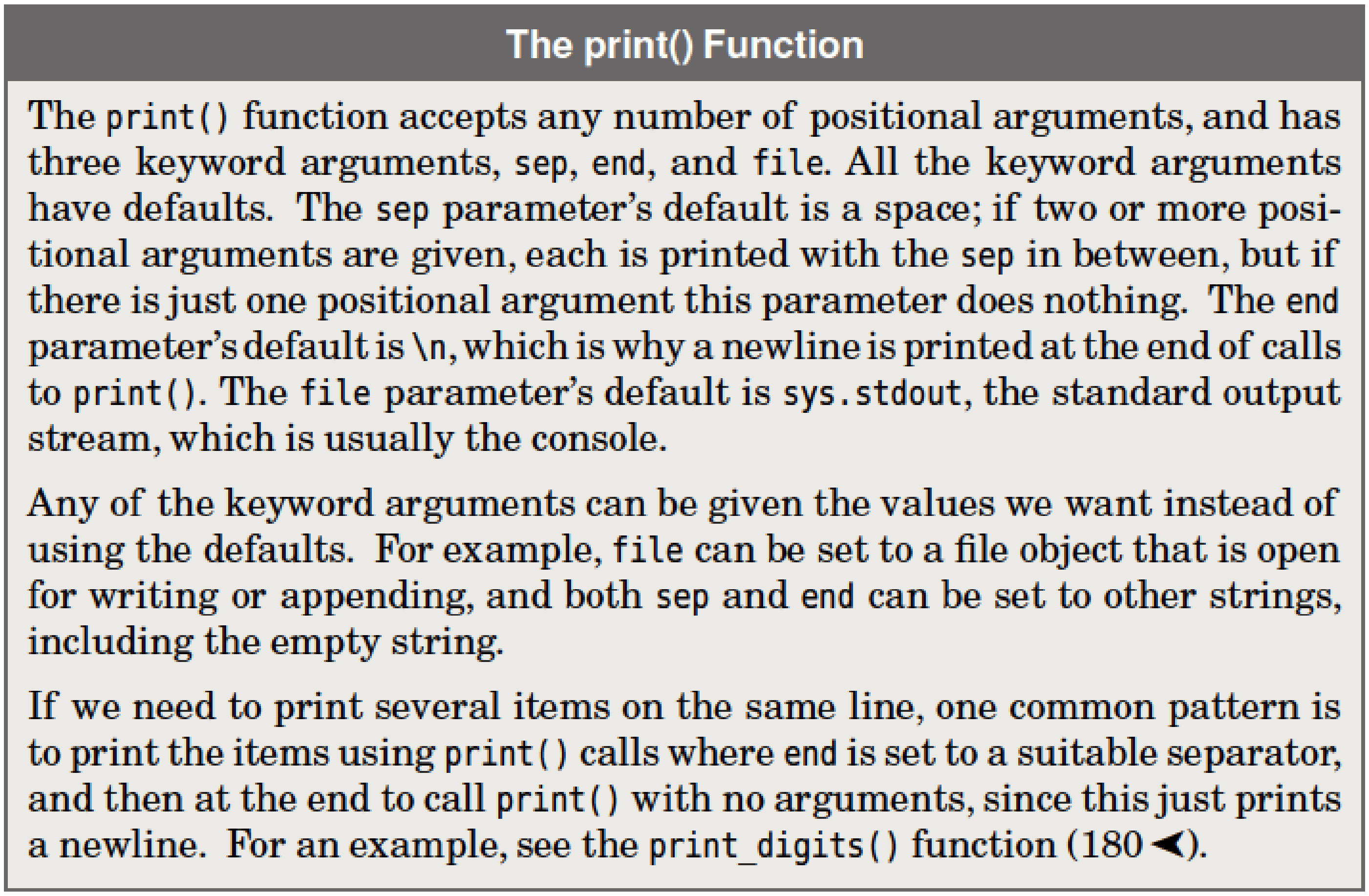
There is a Unicode encoding that uses four bytes per character. It's called UTF-32, because 32 bits = 4 bytes. UTF-32 is a straightforward encoding; it takes each Unicode character (a 4-byte number) and represents the character with that same number. This has some advantages, the most important being that you can find the Nth character of a string in constant time, because the Nth character starts at the 4×Nth byte. It also has several disadvantages, the most obvious being that it takes four freaking bytes to store every freaking character.

Even though there are a lot of Unicode characters, it turns out that most people will never use anything beyond the first 65535.
Thus, there is another Unicode encoding, called UTF-16 (because 16 bits = 2 bytes).
UTF-16 encodes every character from 0–65535 as two bytes, then uses some dirty hacks if you actually need to represent the rarely-used "astral plane" Unicode characters beyond 65535. Most obvious advantage:
UTF-16 is twice as space-efficient as UTF-32, because every character requires only two bytes to store instead of four bytes (except for the ones that don't). And you can still easily find the Nth character of a string in constant time, if you assume that the string doesn't include any astral plane characters, which is a good assumption right up until the moment that it's not.
UTF-8 is a *variable-length* encoding system for Unicode. That is, different characters take up a different number of bytes. For A S C I I characters (A-Z, &c.) U T F - 8 uses just one byte per character. In fact, it uses the exact same bytes; the first 128 characters (0–127) in U T F - 8 are indistinguishable from A S C I I . "Extended Latin" characters like ñ and ö end up taking two bytes. (The bytes are not simply the Unicode code point like they would be in UTF-16; there is some serious bit-twiddling involved.) Chinese characters like 中end up taking three bytes. The rarely-used "astral plane" characters take four bytes.

In Python 3, **all strings are sequences of Unicode characters**. There is no such thing as a Python string encoded in U T F - 8 , or a Python string encoded as CP-1252. "Is this string U T F - 8 ?" is an invalid question.

U T F - 8 is a way of encoding characters as a sequence of bytes. If you want to take a string and turn it into a sequence of bytes in a particular character encoding, Python 3 can help you with that. If you want to take a sequence of bytes and turn it into a string, Python 3 can help you with that too. Bytes are not characters; bytes are bytes. Characters are an abstraction. A string is a sequence of those abstractions.

STRINGS



The print() Function

The `print()` function accepts any number of positional arguments, and has three keyword arguments, `sep`, `end`, and `file`. All the keyword arguments have defaults. The `sep` parameter's default is a space; if two or more positional arguments are given, each is printed with the `sep` in between, but if there is just one positional argument this parameter does nothing. The `end` parameter's default is `\n`, which is why a newline is printed at the end of calls to `print()`. The `file` parameter's default is `sys.stdout`, the standard output stream, which is usually the console.

Any of the keyword arguments can be given the values we want instead of using the defaults. For example, `file` can be set to a file object that is open for writing or appending, and both `sep` and `end` can be set to other strings, including the empty string.

If we need to print several items on the same line, one common pattern is to print the items using `print()` calls where `end` is set to a suitable separator, and then at the end to call `print()` with no arguments, since this just prints a newline. For an example, see the `print_digits()` function (180 ◀).

This will change the separator between strings.

```
> > > print(‘This is a string’, ‘using a single quote!’, sep=’--’)
```

This will print the 2 strings on the same line.

```
> > > print(‘This is a string using a single quote!’ end=’’)
> > > print(‘This is a string using a single quote!’)
```

STRINGS

INPUT AND OUTPUT FUNCTIONS

Output with the print Function

The print function expects zero or more arguments. It evaluates these arguments and converts their values to strings, builds a new string from them with single spaces between them, and displays this string as a single line of text. A line break is automatically displayed as well.

```
>>> print()
>>> print("Hello there!")
Hello there!
>>> print("Text followed by a blank line\n")
Text followed by a blank line
>>> print("Four uses the digit", 4)
Four uses the digit 4
```

Note that print has the effect of stripping off the quotes enclosing each string.

Input with the input Function

The input function expects an optional string as an argument. If the string is provided, the function displays it as a prompt for an input value and waits for the user to enter some keystrokes. When the user presses the Return or Enter key, the function builds and returns a string containing these keyboard characters. The next shell session shows some uses of the input function:

```
>>> input("Enter the radius: ")
Enter the radius: 7.55
'7.55'
>>> input("Press return to quit: ")
Press return to quit: ''
>>> name = input("Enter your name: ")
Enter your name: Ken Lambert
>>> name
'Ken Lambert'
```

When the user enters a string of digits for a number, the programmer must run the **int** or **float** function on this string before using the number in further computations. Here is an example, which inputs the length and width of a rectangle and outputs its area:

```
>>> width = int(input("Enter the width: "))
Enter the width: 34
>>> length = int(input("Enter the length: "))
Enter the length: 22
>>> print("The area is", width * length)
The area is 748
```

STRINGS

STRING BASICS

From a functional perspective, strings can be used to represent just about anything that can be encoded as text or bytes. In the text department, this includes symbols and words (e.g., your name), contents of text files loaded into memory, Internet addresses, Python source code, and so on. Python 3.X comes with three string object types—one for textual data and
two for binary data:

- str for representing decoded Unicode text (including ASCII)
- bytes for representing binary data (including encoded text)
- bytearray, a mutable flavor of the bytes type

UNDERSTANDING DIFFERENT QUOTES

```
> > > print('This is a string using a single quote!')
This is a string using a single quote!
> > > print("This is a string using a double quote!")
This is a string using a double quote!
> > > print("""This string has three quotes!
                Look at what it can do!""")
This string has three quotes
Look at what it can do!
```

CREATING A STRING:

1. **QUOTES:** A string is created by enclosing text in quotes. You can use either single quotes, ', or double quotes, ". A triple-quote can be used for multi-line strings. Here are some examples:

```
> > >  s = 'Hello'
> > >  m = """This is a long string that is
spread across two lines."""
```

2. **INPUT:** Input function

```
>>>s = input('Enter a string: ')
```

3. **str()**

```
>>>s=str()
```

4. **EMPTY STRING**

The empty string '' is the string equivalent of the number 0. It is a string with nothing in it.

```
> > >  s = ''
```

5. **CONCATINATE:** Join strings with the '+' operator

```
> > >  s = 'Hello' +'World'
```

6. **REPLICATE:** If we want to print a long row of dashes, we can do the following

```
>>>print('-'*75)
```

ESCAPE SEQUENCES:

Backslashes are used to introduce special character codings known as *escape sequences*. Escape sequences let us embed characters in strings that cannot easily be typed on a keyboard. The character \, and one or more characters following it in the string literal, are replaced with a *single* character in the resulting string object, which has the binary value specified by the escape sequence. For example, here is a five-character string that embeds a newline and a tab:

```
>>> s = 'a\nb\tc'
```

The two characters \n stand for a single character—the binary value of the newline character in your character set (in ASCII, character code 10). Similarly, the sequence \t is replaced with the tab character. The way this string looks when printed depends on how you print it. The interactive echo shows the special characters as escapes, but print interprets them instead:

```
>>> s
'a\nb\tc'
>>> print(s)
a
b	c
```

To be completely sure how many actual characters are in this string, use the built-in **len** function—it returns the actual number of characters in a string, regardless of how it is coded or displayed:

```
>>> len(s)
5
```

RAW STRINGS SUPPRESS ESCAPES

As we've seen, escape sequences are handy for embedding special character codes within strings. Sometimes, though, the special treatment of backslashes for introducing escapes can lead to trouble, so add the letter *r* to treat the string as a raw string. Strings can also be used to hold the raw bytes used for media files and network transfers, and both the encoded and decoded forms of non- ASCII Unicode text used in internationalized programs. You can place an **r** before the beginning quotation mark of a string to make it a raw string. A raw string completely ignores all escape characters and prints any backslash that appears in the string. For example, type the following into the interactive shell:

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character.

BINARY STRINGS – BYTES

In 3.X all the current string literal forms—'xxx', "xxx", and triplequoted blocks—generate a str; adding a **b** or **B** just before any of them creates a bytes instead. This new **b'...'** bytes literal is similar in form to the **r'...'** raw string used to suppress backslash escapes. **b** or **B** prefix also works for any string literal form, including triple-quoted blocks, though you get back a string of raw bytes that may or may not map to characters. The bytes object is also immutable, just like str.

```
>>> B = b'spam'              # 3.X bytes literal make a bytes object (8-bit bytes)
>>> S = 'eggs'               # 3.X str literal makes a Unicode text string
>>> type(B), type(S)
(<class 'bytes'>, <class 'str'>)
>>> B                # bytes: sequence of int, prints as character string
b'spam'
>>> S
'eggs'
>>> B[0], S[0]               # Indexing returns an int for bytes, str for str
(115, 'e')
>>> B[1:], S[1:]             # Slicing makes another bytes or str object
(b'pam', 'ggs')
>>> B[0] = 'x'       # Both are immutable
TypeError: 'bytes' object does not support item assignment
>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment
```

A function that expects an argument to be a str object won't generally accept a bytes, and vice versa. Because of this, Python 3.X basically requires that you commit to one type or the other,
or perform manual, explicit conversions when needed:

- str.encode() and bytes(S, encoding) translate a string to its raw bytes form and create an encoded bytes from a decoded str in the process.
- bytes.decode() and str(B, encoding) translate raw bytes into its string form and create a decoded str from an encoded bytes in the process.

```
>>> S = 'eggs'
>>> S.encode()                          # str->bytes: encode text into raw bytes
b'eggs'
>>> bytes(S, encoding='ascii')          # str->bytes, alternative
b'eggs'
>>> B = b'spam'
>>> B.decode()                          # bytes->str: decode raw bytes into text
'spam'
>>> str(B, encoding='ascii')            # bytes->str, alternative
'spam'
```

STRINGS

EXPRESSION OPERATIONS:

Unlike in C, in Python, strings come with a powerful set of processing tools. Also unlike languages such as C, Python has no distinct type for individual characters; instead, you just use one-character strings.

For processing, strings support *expression* operations such as concatenation (combining strings), slicing (extracting sections), indexing (fetching by offset), and so on. Besides expressions, Python also provides a set of string *methods* that implement common string-specific tasks, as well as *modules* for more advanced text-processing tasks such as pattern matching.

String Special Operators: Assume string variable a holds 'Hello' and variable b holds 'Python', then-

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 print(R'\n') prints \n
%	Format- Performs String formatting – acts as a placeholder	

STRINGS

ACCESSING VALUES IN STRINGS

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example-

```
var1 = 'Hello World!'
var2 = "Python Programming"
print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
```

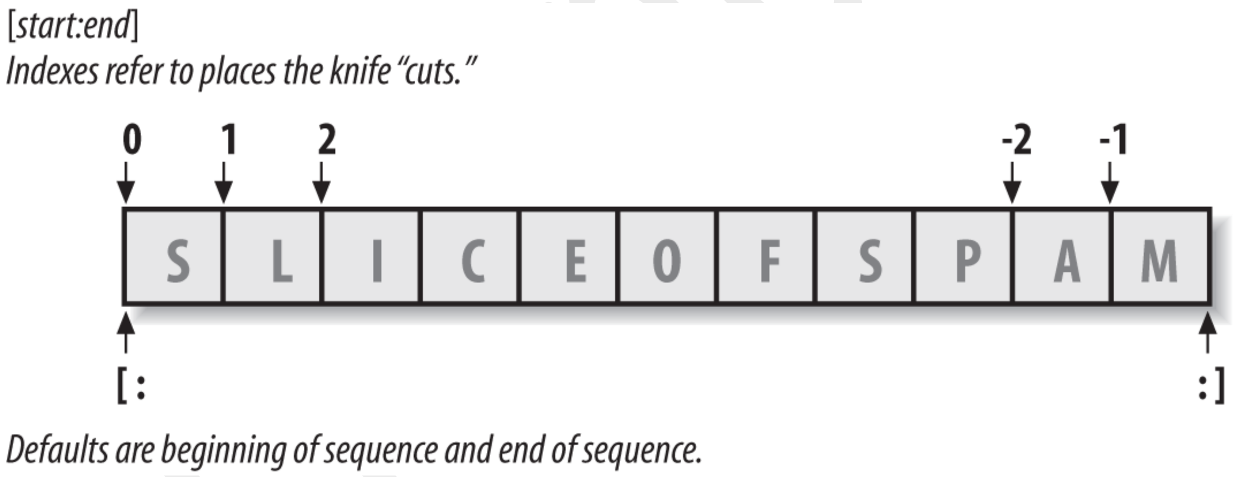
When the above code is executed, it produces the following result

```
var1[0]: H
var2[1:5]: ytho
```

INDEXING & SLICING

The most basic string access operation is indexing or subscripting. This operation gives you the character at a given position in the string. Each character in a string has a definite position. These positions range from zero, the position of the first character, to the length of the string minus one, the position of the last character.

The form of this operation is as follows: aString[anIntegerIndexPosition].



[start:end]
Indexes refer to places the knife "cuts."
0
1
2
-2
-1
S L I C E O F S P A M
[:
:]
Defaults are beginning of sequence and end of sequence.

The integer in square brackets following a string is also called a subscript, and the [ ] operator is called the subscript operator. Here is a shell session with some example subscript operations:

```
>>> name = "Ken Lambert"
>>> name
'Ken Lambert'
>>> name[0]
'K'
>>> name[len(name) - 1]
't'
>>> name[len(name)]
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
name[len(name)]
IndexError: string index out of range
```

The error on the last line results from giving an index position that lies beyond the sequence of positions in the string.

Negative index values are allowed, counting from –1 (the last position) down to minus the length of the string (the first position). You may copy some characters from String data. To do it, you can use [start:end] syntax.

Here is syntax format: use it in a file and execute. If you specify an index, you'll get the character at that position in the string. If you specify a range from one index to another, the starting index is included and the ending index is not.

```
msg = 'Hello world, Python!'
# copy / Slice
print(msg[5:])
print(msg[:5])
print(msg[-3:])
print(msg[:-3])
print(msg[2:6])
print(msg[5:8])
```

Here's a summary of the details for reference:

Indexing (S[i]) fetches components at offsets:
• The first item is at offset 0.
• Negative indexes mean to count backward from the end or right.
• S[0] fetches the first item.
• S[−2] fetches the second item from the end (like S[len(S)−2]).

Slicing (S[i:j]) extracts contiguous sections of sequences:
• The upper bound is non-inclusive.
• Slice boundaries default to 0 and the sequence length, if omitted.
• S[1:3] fetches items at offsets 1 up to but not including 3.
• S[1:] fetches items at offset 1 through the end (the sequence length).
• S[:3] fetches items at offset 0 up to but not including 3.
• S[:−1] fetches items at offset 0 up to but not including the last item.
• S[:] fetches items at offsets 0 through the end—making a top-level copy of S.

Extended slicing (S[i:j:k]) accepts a step (or stride) k, which defaults to +1:
• Allows for skipping items and reversing order

slice expressions have support for an optional third index, used as a *step* (sometimes called a *stride*). The step is added to the index of each item extracted.

The full-blown form of a slice is now X[*I*:*J*:*K*], which means "extract all the items in X, from offset *I* through *J*−1, by *K*." The third limit, *K*, defaults to +1, which is why normally all items in a slice are extracted from left to right. If you specify an explicit value, however, you can use the third limit to skip items or to reverse their order.

For instance, X[1:10:2] will fetch *every other item* in X from offsets 1–9; that is, it will collect the items at offsets 1, 3, 5, 7, and 9. As usual, the first and second limits default to 0 and the length of the sequence, respectively, so X[::2] gets every other item from the beginning to the end of the sequence:

```
>>> S = 'abcdefghijklmnop'
>>> S[1:10:2] # Skipping items
'bdfhj'
>>> S[::2]
'acegikmo'
```

You can also use a negative stride to collect items in the opposite order. For example, the slicing expression "hello"[::−1] returns the new string "olleh"—the first two bounds default to 0 and the length of the sequence, as before, and a stride of −1 indicates that the slice should go from right to left instead of the usual left to right. The effect, therefore, is to *reverse* the sequence:

```
>>> S = 'hello'
>>> S[::−1]            # Reversing items
'olleh'
```

With a negative stride, the meanings of the first two bounds are essentially reversed.
That is, the slice S[5:1:−1] fetches the items from 2 to 5, in reverse order (the result contains items from offsets 5, 4, 3, and 2):

```
>>> S = 'abcedfg'
>>> S[5:1:−1]                # Bounds roles differ
'fdec'
```

slicing is equivalent to indexing with a *slice object*,

```
>>> 'spam'[1:3]              # Slicing syntax
'pa'
>>> 'spam'[slice(1, 3)]      # Slice objects with index syntax + object
'pa'
>>> 'spam'[::-1]
'maps'
>>> 'spam'[slice(None, None, −1)]
'maps'
```

UPDATING STRINGS

Python strings are categorized as ***immutable sequences***, meaning that the characters they contain have a left-to-right positional order and that they cannot be changed in place.

```
>>> S = 'spam'
>>> S[0] = 'x'               # Raises an error!
TypeError: 'str' object does not support item assignment
```

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.
For example-

```
var1 = 'Hello World!'
print ("Updated String :- ", var1[:6] + 'Python')
```

When the above code is executed, it produces the following result-

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

STRINGS

STRING CONCATENATION AND REPLICATION

The + operator. The meaning of an operator may change based on the data types of the values next to it. For example, + is the addition operator when it operates on two integers or floating-point values. However, when + is used on two string values, it joins the strings as the string concatenation operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the + operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.

```
>>> 'Alice' + 42
Traceback (most recent call last):
File "<pyshell#26>", line 1, in <module>
'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

The * operator, is used for multiplication when it operates on two integer or floating-point values. But when the * operator is used on one string value and one integer value, it becomes the string replication operator. Enter a string multiplied by a number into the interactive shell to see this in action.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

The * operator can be used with only two numeric values (for multiplication) or one string value and one integer value (for string replication). Otherwise, Python will just display an error message.

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
```

```
>>> 'Alice' * 5.0
Traceback (most recent call last):
File "<pyshell#33>", line 1, in <module>
'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

JOINING STRINGS WITH THE FORMAT() FUNCTION

```
>>> print("{} {}".format(str1, str2))
>>> template = '{0}, {1} and {2}'            # By position
>>> template.format('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '{a}, {b} and {c}'            # By keyword
>>> template.format(a='spam', b='ham', c='eggs')
'spam, ham and eggs'

>>> template = '{a}, {0} and {c}'            # By both
>>> template.format('ham', a='spam', c='eggs')
'spam, ham and eggs'

>>> template = '{}, {} and {}'               # By relative position
>>> template.format('spam', 'ham', 'eggs')   # New in 3.1 and 2.7
'spam, ham and eggs'
```

THE 'IN' AND 'NOT IN' OPERATORS WITH STRINGS

The in and not in operators can be used with strings just like with list values. An expression with two strings joined using in or not in will *evaluate to a Boolean True or False*. Enter the following into the interactive shell:

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

CONVERSION.

String To Numeric

Sometime you want to do math operations but input data has string type. To convert string type into numeric, you can use int() for String to Integer and float() for string to Float. Try the following sample code to implement string to numeric conversion.

```
# string to numeric
> > > a = "2"
> > > b = "6.8"
> > > num1 = int(a)
> > > num2 = float(b)
> > > print(num1)
> > > print(num2)
```

Numeric to String

It is easy to convert numeric to String type, you can use str(). You can get string type automatically.

```
# numeric to string
> > > a = 6
> > > b = 8.56
> > > str1 = str(a)
> > > str2 = str(b)
> > > print(str1)
> > > print(str2)
```

STRINGS

STRING PARSER

The simple solution to parsing String uses split() with delimiter parameter. For example, you have String data with ‘;’ delimiter and want to parse it. Here is sample code

```
> > > msg = 'Berlin;Amsterdam;London;Tokyo'
> > > cities = msg.split(';')
>>> cities
['Berlin', 'Amsterdam', 'London', 'Tokyo']
```

FUNCTIONS USED WITH STRINGS

DIR:

When you run dir(str) in the shell, you see all the string operations. They are called methods. Note that the method names at the beginning of this list contain underscores. These methods are associated with operators such as + and []. When Python sees an operator used with a string, it looks up the corresponding method and calls it with the appropriate arguments.

```
>>> a= 'some string'
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

LENGTH

Check String Data Length: You can use len() to get the length of data.

```
>>> a= 'some string'
>>> len(a)
11
>>>
```

STRINGS

BUILT-IN STRING METHODS

Python includes the following built-in methods to manipulate strings Methods with Description

capitalize()

Capitalizes first letter of string

```
str = "this is string example....wow!!!"
print ("str.capitalize() : ", str.capitalize())
```

Result

```
str.capitalize() : This is string example....wow!!!
```

center(width, fillchar)

Returns a string padded with fillchar with the original string centered to a total of width columns.

```
width - This is the total width of the string.
fillchar - This is the filler character.
str = "this is string example....wow!!!"
print ("str.center(40, 'a') : ", str.center(40, 'a'))
```

Result

```
str.center(40, 'a') : aaaathis is string example....wow!!!aaaa
```

count(str, beg= 0,end=len(string))

Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.

sub - This is the substring to be searched.
start - Search starts from this index.
end - Search ends from this index.

```
str="this is string example....wow!!!"
sub='i'
print ("str.count('i') : ", str.count(sub))
sub='exam'
print ("str.count('exam', 10, 40) : ", str.count(sub,10,40))
```

Result

```
str.count('i') : 3
str.count('exam', 4, 40) : 1
```

endswith(suffix, beg=0, end=len(string))

Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.

suffix - This could be a string or could also be a tuple of suffixes to look for.
start - The slice begins from here.
end - The slice ends here.

```
Str='this is string example....wow!!!'
suffix='!!'
print (Str.endswith(suffix))
print (Str.endswith(suffix,20))
suffix='exam'
print (Str.endswith(suffix))
print (Str.endswith(suffix, 0, 19))
```

Result

```
True
True
False
True
```

find(str, beg=0 end=len(string))
Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.

str - This specifies the string to be searched.
beg - This is the starting index, by default its 0.
end - This is the ending index, by default its equal to the length of the string.

```
str1 = "this is string example....wow!!!"
str2 = "exam";
print (str1.find(str2))
print (str1.find(str2, 10))
print (str1.find(str2, 40))
```

Result

```
15
15
-1
```

index(str, beg=0, end=len(string))
Same as find(), but raises an exception if str not found.

str - This specifies the string to be searched.
beg - This is the starting index, by default its 0.
end - This is the ending index, by default its equal to the length of the string.

```
str1 = "this is string example....wow!!!"
str2 = "exam";
print (str1.index(str2))
print (str1.index(str2, 10))
print (str1.index(str2, 40))
```

Result

```
15
15
Traceback (most recent call last):
File "test.py", line 7, in
print (str1.index(str2, 40))
ValueError: substring not found
shell returned 1
```

isalnum()
Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

```
str = "this2016" # No space in this string
print (str.isalnum())
str = "this is string example....wow!!!"
print (str.isalnum())
```

Result

```
True
False
```

isalpha()
Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

```
str = "this"; # No space & digit in this string;print (str.isalpha())
str = "this is string example....wow!!!";print (str.isalpha())
```

Result

```
True
False
```

isdigit()
Returns true if the string contains only digits and false otherwise.

```
str = "123456"; # Only digit in this string
print (str.isdigit())
str = "this is string example....wow!!!"
print (str.isdigit())
```

Result

```
True
False
```

islower()
Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.

```
str = "THIS is string example....wow!!!"
print (str.islower())
str = "this is string example....wow!!!"
print (str.islower())
```

Result

```
False
True
```

isnumeric()
Returns true if a unicode string contains only numeric characters and false otherwise.

```
str = "this2016"
print (str.isnumeric())
str = "23443434"
print (str.isnumeric())
```

Result

```
False
True
```

isspace()
Returns true if string contains only whitespace characters and false otherwise.

```
str = " "
print (str.isspace())
str = "This is string example....wow!!!"
print (str.isspace())
```

Result

```
True
False
```

istitle()
Returns true if string is properly "titlecased" and false otherwise.

```
str = "This Is String Example...Wow!!!"
print (str.istitle())
str = "This is string example....wow!!!"
print (str.istitle())
```

Result

```
True
False
```

join(seq)
Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.

sequence - This is a sequence of the elements to be joined.

```
s = "-"
seq = ("a", "b", "c")           # This is sequence of strings.
print (s.join( seq ))
```

Result

```
a-b-c
```

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

ljust(width[, fillchar])
Returns a space-padded string with the original string left-justified to a total of width columns.

width - This is string length in total after padding.
fillchar - This is filler character, default is a space.

```
str = "this is string example....wow!!!"
print str.ljust(50, '*')
```

Result

```
this is string example....wow!!!******************
```

lower()
Converts all uppercase letters in string to lowercase.

```
str = "THIS IS STRING EXAMPLE....WOW!!!"
print (str.lower())
```

Result

```
this is string example....wow!!!
```

lstrip()
Removes all leading whitespace in string.

chars - You can supply what chars have to be trimmed.

```
str = " this is string example....wow!!!"
print (str.lstrip())
str = "*****this is string example....wow!!!*****"
print (str.lstrip('*'))
```

Result

```
this is string example....wow!!!
this is string example....wow!!!*****
```

max(str)
Returns the max alphabetical character from the string str.

str - This is the string from which max alphabetical character needs to be returned.

```
str = "this is a string example....really!!!"
print ("Max character: " + max(str))
str = "this is a string example....wow!!!"
print ("Max character: " + max(str))
```

Result

```
Max character: y
Max character: x
```

min(str)
Returns the min alphabetical character from the string str.

str - This is the string from which min alphabetical character needs to be returned.

```
str = "www.tutorialspoint.com"
print ("Min character: " + min(str))
str = "TUTORIALSPOINT"
print ("Min character: " + min(str))
```

Result

```
Min character: .
Min character: A
```

replace(old, new [, max])
Replaces all occurrences of old in string with new or at most max occurrences if max given.

old - This is old substring to be replaced.
new - This is new substring, which would replace old substring.
max - If this optional argument max is given, only the first count occurrences are replaced.

```
str = "this is string example....wow!!! this is really string"
print (str.replace("is", "was"))
print (str.replace("is", "was", 1))
```

Result

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

rfind(str, beg=0,end=len(string))
Same as find(), but search backwards in string.

str - This specifies the string to be searched.
beg - This is the starting index, by default its 0.
end - This is the ending index, by default its equal to the length of the string.

```
str1 = "this is really a string example....wow!!!"
str2 = "is"
print (str1.rfind(str2))
print (str1.rfind(str2, 0, 10))
print (str1.rfind(str2, 10, 0))
print (str1.find(str2))
print (str1.find(str2, 0, 10))
print (str1.find(str2, 10, 0))
```

Result

```
5
5
-1
2
2
-1
```

rindex(str, beg=0, end=len(string))
Same as index(), but search backwards in string.

str - This specifies the string to be searched.
beg - This is the starting index, by default its 0.
len - This is ending index, by default its equal to the length of the string.

```
str1 = "this is really a string example....wow!!!"
str2 = "is"
print (str1.rindex(str2))
print (str1.rindex(str2,10))
```

Result

```
5
Traceback (most recent call last):
File "test.py", line 5, in
print (str1.rindex(str2,10))
ValueError: substring not found
```

rjust(width,[, fillchar])
Returns a space-padded string with the original string right-justified to a total of width columns.

width - This is the string length in total after padding.
fillchar - This is the filler character, default is a space.

```
str = "this is string example....wow!!!"
print (str.rjust(50, '*'))
```

Result

```
******************this is string example....wow!!!
```

rstrip()
Removes all trailing whitespace of string.

chars - You can supply what chars have to be trimmed.

```
str = " this is string example....wow!!! "
print (str.rstrip())
str = "*****this is string example....wow!!!*****"
print (str.rstrip('*'))
```

Result

```
this is string example....wow!!!
*****this is string example....wow!!!
```

split(str="", num=string.count(str))
Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.

str - This is any delimeter, by default it is space.
num - this is number of lines to be made

```
str = "this is string example....wow!!!"
print (str.split( ))
print (str.split('i',1))
print (str.split('w'))
```

Result

```
['this', 'is', 'string', 'example....wow!!!']
['th', 's is string example....wow!!!']
['this is string example....', 'o', '!!!']
```

splitlines(num=string.count('\n'))
Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.

num - This is any number,

```
str = "this is \nstring example....\nwow!!!"
print (str.splitlines( ))
```

Result

```
['this is ', 'string example....', 'wow!!!']
```

startswith(str, beg=0,end=len(string))
Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

str - This is the string to be checked.
beg - This is the optional parameter to set start index of the matching boundary.
end - This is the optional parameter to set start index of the matching boundary.

```
str = "this is string example....wow!!!"
print (str.startswith( 'this' ))
print (str.startswith( 'string', 8 ))
print (str.startswith( 'this', 2, 4 ))
```

Result

```
True
True
False
```

strip([chars])
Performs both lstrip() and rstrip() on string

chars - The characters to be removed from beginning or end of the string.

```
str = "*****this is string example....wow!!!*****"
print (str.strip( '*' ))
```

Result

```
this is string example....wow!!!
```

swapcase()
Inverts case for all letters in string.

```
str = "this is string example....wow!!!"
print (str.swapcase())
str = "This Is String Example....WOW!!!"
print (str.swapcase())
```

Result

```
THIS IS STRING EXAMPLE....WOW!!!
tHIS iS sTRING eXAMPLE....wow!!!
```

title()
Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.

```
str = "this is string example....wow!!!"
print (str.title())
```

Result

```
This Is String Example....Wow!!!
```

upper()
Converts lowercase letters in string to uppercase.

```
str = "this is string example....wow!!!"
print ("str.upper : ",str.upper())
```

Result

```
str.upper : THIS IS STRING EXAMPLE....WOW!!!
```

isdecimal()
Returns true if a unicode string contains only decimal characters and false otherwise.

```
str = "this2016"
print (str.isdecimal())
str = "23443434"
print (str.isdecimal())
```

Result

```
False
True
```

STRINGS

QUICK REFERENCE:

Operation	Interpretation
S = ''	Empty string
S = "spam's"	Double quotes, same as single
S = 's\np\ta\x00m'	Escape sequences
S = """*...multiline...*"""	Triple-quoted block strings
S = r'\temp\spam'	Raw strings (no escapes)
B = b'sp\xc4m'	Byte strings in 2.6, 2.7, and 3.X
U = u'sp\u00c4m'	Unicode strings in 2.X and 3.3+
S1 + S2	Concatenate
S * 3	repeat
S[i]	Index
S[i:j]	Slice
len(S)	length
"a %s parrot" % kind	String formatting expression
"a {0} parrot".format(kind)	String formatting method in 2.6, 2.7, and 3.X
S.find('pa')	String methods - search
S.rstrip()	remove whitespace
S.replace('pa', 'xx')	replacement
S.split(',')	split on delimiter
S.isdigit()	content test
S.lower()	case conversion
S.endswith('spam')	end test
'spam'.join(strlist)	delimiter join
S.encode('latin-1')	Unicode encoding
B.decode('utf8')	Unicode decoding
for x in S: print(x)	Iteration
'spam' in S	membership
[c * 2 for c in S]	Comprehension
map(ord, S)	map
re.match('sp(.*)am', line)	Pattern matching: library module

Python also supports more advanced pattern-based string processing with the standard library's re (for "regular expression") module and even higher-level text processing tools such as XML parsers