Python 3 Notes

Table of Contents

Python 3 Notes

Python Shell (IDLE) Window:

Control-c interrupts executing command.

Control-d sends end-of-file; closes window if typed at >>> prompt.

Command history:

Alt-p retrieves previous command matching what you have typed.

Alt-n retrieves next.

(These are Control-p, Control-n on OS X)

Return while cursor is on a previous command retrieves that command.

Expand word is also useful to reduce typing.

Syntax colors:

The coloring is applied in a background "thread", so you may occasionally see uncolorized text. To change the color scheme, use the Configure IDLE / Highlighting dialog.

Python default syntax colors:

Keywords	orange
Builtins	royal purple
Strings	green
Comments	red
Definitions	blue

Shell default colors:

Console output	brown
stdout	blue
stderr	red
stdin	black

Python 3.5 Built-in Functions

The Python interpreter has a number of functions built into it that are always available. No Import is required to access them. They are listed here in alphabetical order.

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Python 3 Notes

Pointers to Good Articles/Sites

To download Python for Linux, Windows or OSX, go to the python.org download page. If you are unable to compile python in a Linux environment, you can get binaries to install at Active State. They have free versions of the latest software for version 2 and version 3. They have installation directions for you to follow. Of course, most Linux distributions come with both versions of Python now. It is possible to do these installations without administrator privileges.

An alternative is to use Miniconda. Do not install the full Anaconda. It is about 4 GB of software, most of which you do not need. This link will take you to a page that includes binaries for Linux, OSX and Windows. It includes options for Python 2 or 3 and it points you to installation instructions. Administrator privileges are not required. In this case, you get to IDLE strictly through the command-line tool provided with Miniconda. Just issue idle or idle3 as a command and IDLE will appear. Then you can minimize the command-line window. Do not shut it down as that will shut down IDLE.

A better choice for production is the Spyder IDE which is simple to install with Miniconda. Just issue "conda install spyder" and all the necessary software will be installed. Be patient. There is quite a bit of software being installed.

A good book for learning Python 3 is included in your data download. It used to be entiled, "Python for Informatics." Now it is entitled, "Python for Everybody." This book avoids the mathematics found in the book, "Think Python" and concentrates on the mechanics of the Python language. It also goes into some of the more interesting aspects of the language. The book, "Think Python" is also included in your data.

Green Tea Press features a number of free books in PDF format. They cover a myriad of topics, not just Python. http://greenteapress.com/

Good sites for learning Python:

- https://www.tutorialspoint.com/python3/index.htm (downloadable as a PDF)
  This site includes a discussion of the fundamental differences between versions 2 and 3. It is not all inclusive.
- http://www.python-course.eu/python3_course.php
  This site has extensive tutorial material as well as intermediate and advanced material. It is very thorough and well-constructed.
- https://automatetheboringstuff.com/ - is a free book which reviews the basics and outlines projects to take on to give you development experience. You should have completed Python II at a minimum before you attack this book.
- Using Python 3.X: http://getpython3.com/diveintopython3/ - this is a more advanced site; not a tutorial.

The philosophy of Python according to Tim Peters – good general guide for producing Python programs.

Python 3 Notes

This article describes hashing as it pertains to dictionaries in relatively simple terms. http://www.i-programmer.info/babbages-bag/479-hashing.html.

A complete summary of the differences between Python 2 and 3 is covered at this site: http://python-future.org/compatible_idioms.html. This site includes a PDF version which you can download.

This site (https://www.tutorialspoint.com/python3/python_exceptions.htm) lists all of the standard errors that can occur and be intercepted with a try statement.

Formatting data into strings:

Older Method:

The following is an abbreviation of the full explanation of formatting that can be found at: http://docs.python.org/2/library/stdtypes.html#string-formatting. The explanation of the percent sign has been expanded. The format (or specifier) itself is a string of characters:

1. The '%' character marks the start of the specifier.
2. Conversion flags (optional), which affect the result of some conversion types.
3. Minimum field width (optional). This is just an integer.
4. Precision (optional), given as a '.' (dot) followed by the precision or number of decimal places.
5. Conversion type.

Conversion Flag Table:

Flag	Meaning
'0'	The conversion will be zero padded for numeric values.
'-'	The converted value is left adjusted (overrides the '0' conversion if both are given).
' '	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
'+'	A sign character ('+' or '-') will precede the conversion (overrides a “space” flag).

The conversion types are:

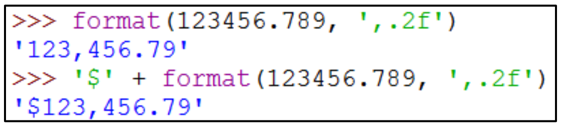
Conversion	Meaning
'd'	Signed integer.
'i'	Signed integer.
'f'	Floating point format.
'F'	Floating point format.
'c'	Single character (accepts integer or single character string).
'r'	String (converts any Python object using *repr()*).
's'	String (converts any Python object using **str()**).
'%%'	No argument is converted, results in a '%' character in the result.

Conversion	Meaning
	Ex: '%.1f%%' % 12.345 gives the string 12.3%

Example:

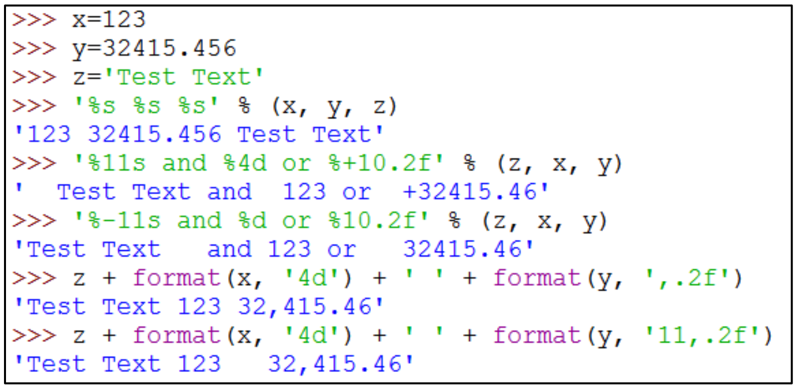
%+-7.1f – this format will force a sign to show up (+), will left justify the entire entry in the width (-), will assign a width of 7 to the result and will cause the number to be displayed as a floating-point number (f). If the number is 12.345, the result will be the string '+12.3 '. The results of all formatting operations are strings.

There is also the format function which is the only way to implement a thousands separator using the method we are learning (e.g., 12345.67 printed as 12,345.67). The format function has two arguments; the number to be formatted and the format to apply. You do not use the % symbol in these formats.
Examples:



```
>>> format(123456.789, ',.2f')
'123,456.79'
>>> '$' + format(123456.789, ',.2f')
'$123,456.79'
```

Some more examples of formatting. Note how the floating point number is rounded.



```
>>> x=123
>>> y=32415.456
>>> z='Test Text'
>>> '%s %s %s' % (x, y, z)
'123 32415.456 Test Text'
>>> '%11s and %4d or %+10.2f' % (z, x, y)
'  Test Text and  123 or  +32415.46'
>>> '%-11s and %d or %10.2f' % (z, x, y)
'Test Text   and 123 or   32415.46'
>>> z + format(x, '4d') + ' ' + format(y, ',.2f')
'Test Text 123 32,415.46'
>>> z + format(x, '4d') + ' ' + format(y, '11,.2f')
'Test Text 123   32,415.46'
```

Newer Method[1]:

General statement: 'text & formatting sequence(s)'.format(variables/literals to be inserted)
The general format of a formatting sequence is:
{[seq#] ":" [[fill] align] [sign] ["#"] ["0"] [width] [","] ["." prec] [type]}
The valid types are s, d and f . (for now. There are many more.)
s – strings, d – integers, f – floating-point numbers
These sequences are used to insert data into a string.
f is usually preceded by a .n – where n is an integer specifying the number of decimal places to display.

[1] The "newer method is no longer the newest method. Python 3.6 introduced f-strings. Understanding the "newer" method here makes learning f-strings very easy.

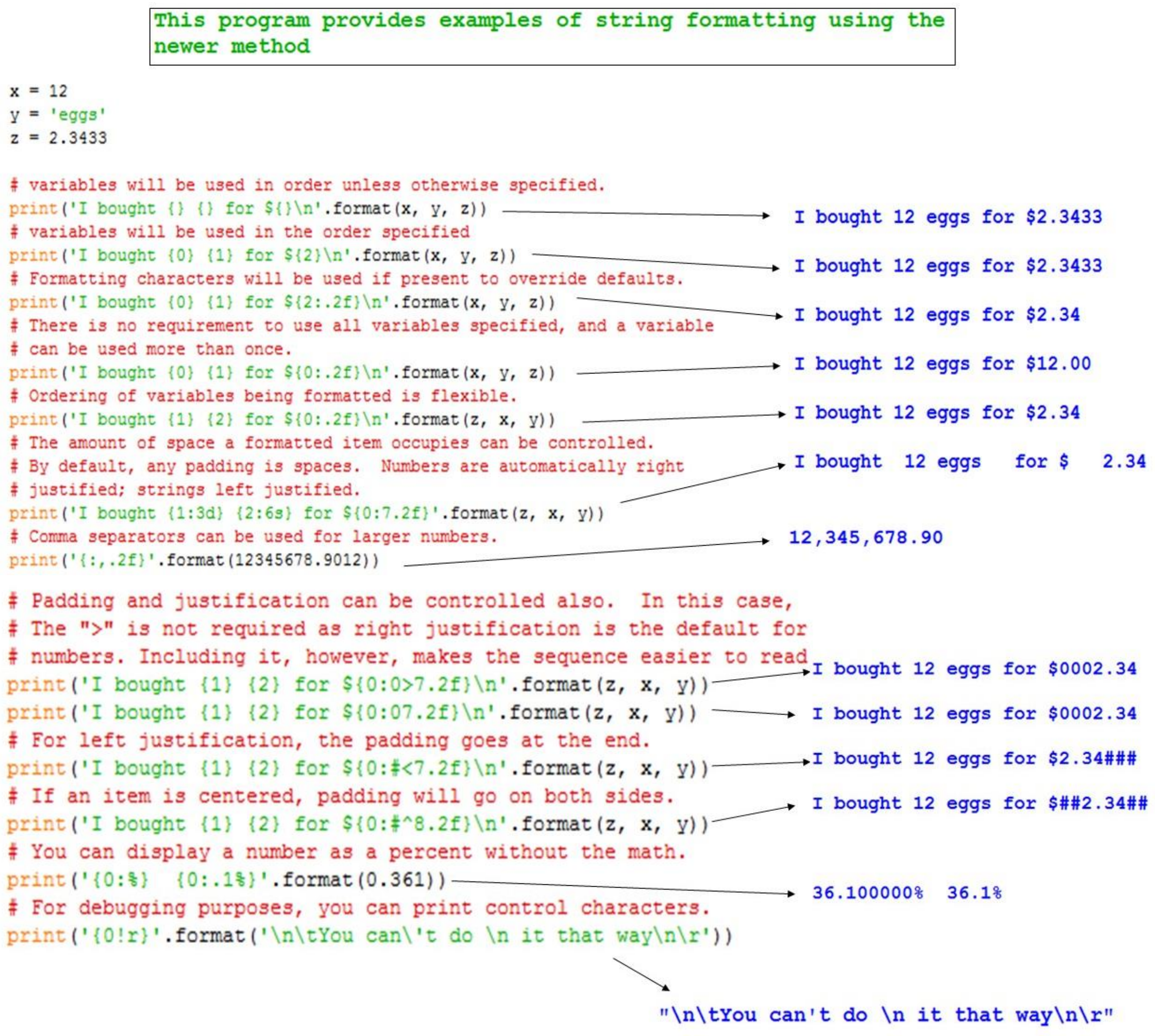
Python 3 Notes

General example:

x = 'Some text {0:5d} more text {1:7.2f}'.format(12, 17.426)

Result stored as x – 'Some text   12 more text   17.43' (Note rounding)

More examples:



```
This program provides examples of string formatting using the
newer method

x = 12
y = 'eggs'
z = 2.3433

# variables will be used in order unless otherwise specified.
print('I bought {} {} for ${}\n'.format(x, y, z))
# variables will be used in the order specified
print('I bought {0} {1} for ${2}\n'.format(x, y, z))
# Formatting characters will be used if present to override defaults.
print('I bought {0} {1} for ${2:.2f}\n'.format(x, y, z))
# There is no requirement to use all variables specified, and a variable
# can be used more than once.
print('I bought {0} {1} for ${0:.2f}\n'.format(x, y, z))
# Ordering of variables being formatted is flexible.
print('I bought {1} {2} for ${0:.2f}\n'.format(z, x, y))
# The amount of space a formatted item occupies can be controlled.
# By default, any padding is spaces.  Numbers are automatically right
# justified; strings left justified.
print('I bought {1:3d} {2:6s} for ${0:7.2f}'.format(z, x, y))
# Comma separators can be used for larger numbers.
print('{:,.2f}'.format(12345678.9012))

# Padding and justification can be controlled also.  In this case,
# The ">" is not required as right justification is the default for
# numbers. Including it, however, makes the sequence easier to read
print('I bought {1} {2} for ${0:0>7.2f}\n'.format(z, x, y))
print('I bought {1} {2} for ${0:07.2f}\n'.format(z, x, y))
# For left justification, the padding goes at the end.
print('I bought {1} {2} for ${0:#<7.2f}\n'.format(z, x, y))
# If an item is centered, padding will go on both sides.
print('I bought {1} {2} for ${0:#^8.2f}\n'.format(z, x, y))
# You can display a number as a percent without the math.
print('{0:%}  {0:.1%}'.format(0.361))
# For debugging purposes, you can print control characters.
print('{0!r}'.format('\n\tYou can\'t do \n it that way\n\r'))
```

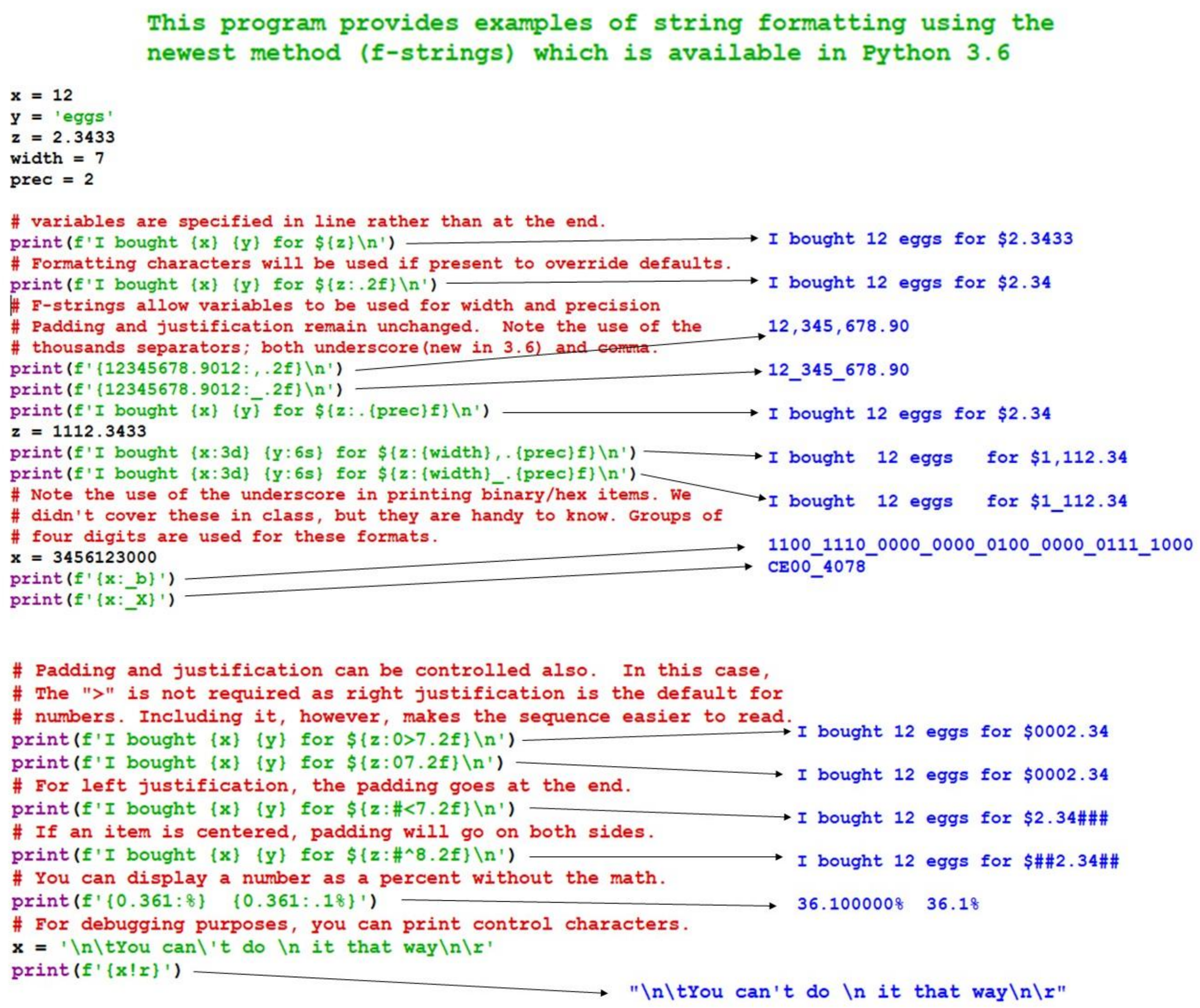
```
I bought 12 eggs for $2.3433
I bought 12 eggs for $2.3433
I bought 12 eggs for $2.34
I bought 12 eggs for $12.00
I bought 12 eggs for $2.34
I bought  12 eggs   for $   2.34
12,345,678.90
I bought 12 eggs for $0002.34
I bought 12 eggs for $0002.34
I bought 12 eggs for $2.34###
I bought 12 eggs for $##2.34##
36.100000%  36.1%
"\n\tYou can't do \n it that way\n\r"
```

For a complete definition of the newer string formatting method, New Mexico Tech has a great web site explaining the whole process in much clearer language than the formal python documentation. This link takes you to the detailed specification portion. If you want a lot more detail, start here although it is probably more information than you want at this point. A relatively simple set of examples can be found on Marcus Kazmierczak's Personal Site.

Python 3 Notes

Newest Method:

The newest formatting capability was introduced in Python 3.6.   It employs f-strings which allow you to specify a variable/literal/expression in line rather than as input to the format method. They also allow a few other capabilities. As before, you put the object of the formatting as well as any formatting directives inside braces. The samples below should give you an idea of how to use f-strings.



```
            This program provides examples of string formatting using the
            newest method (f-strings) which is available in Python 3.6

x = 12
y = 'eggs'
z = 2.3433
width = 7
prec = 2

# variables are specified in line rather than at the end.
print(f'I bought {x} {y} for ${z}\n')
# Formatting characters will be used if present to override defaults.
print(f'I bought {x} {y} for ${z:.2f}\n')
# F-strings allow variables to be used for width and precision
# Padding and justification remain unchanged.  Note the use of the
# thousands separators; both underscore(new in 3.6) and comma.
print(f'{12345678.9012:,.2f}\n')
print(f'{12345678.9012:_.2f}\n')
print(f'I bought {x} {y} for ${z:.{prec}f}\n')
z = 1112.3433
print(f'I bought {x:3d} {y:6s} for ${z:{width},.{prec}f}\n')
print(f'I bought {x:3d} {y:6s} for ${z:{width}_.{prec}f}\n')
# Note the use of the underscore in printing binary/hex items. We
# didn't cover these in class, but they are handy to know. Groups of
# four digits are used for these formats.
x = 3456123000
print(f'{x:_b}')
print(f'{x:_X}')


# Padding and justification can be controlled also.  In this case,
# The ">" is not required as right justification is the default for
# numbers. Including it, however, makes the sequence easier to read.
print(f'I bought {x} {y} for ${z:0>7.2f}\n')
print(f'I bought {x} {y} for ${z:07.2f}\n')
# For left justification, the padding goes at the end.
print(f'I bought {x} {y} for ${z:#<7.2f}\n')
# If an item is centered, padding will go on both sides.
print(f'I bought {x} {y} for ${z:#^8.2f}\n')
# You can display a number as a percent without the math.
print(f'{0.361:%}  {0.361:.1%}')
# For debugging purposes, you can print control characters.
x = '\n\tYou can\'t do \n it that way\n\r'
print(f'{x!r}')
```

```
I bought 12 eggs for $2.3433
I bought 12 eggs for $2.34
12,345,678.90
12_345_678.90
I bought 12 eggs for $2.34
I bought  12 eggs   for $1,112.34
I bought  12 eggs   for $1_112.34
1100_1110_0000_0000_0100_0000_0111_1000
CE00_4078

I bought 12 eggs for $0002.34
I bought 12 eggs for $0002.34
I bought 12 eggs for $2.34###
I bought 12 eggs for $##2.34##
36.100000%  36.1%
"\n\tYou can't do \n it that way\n\r"
```

A good online view of f-strings can be found here. Pay special attention to the multi-line capability. Also, I have seen claims that f-strings are much more efficient than any of the other formatting capabilities in Python. In certain cases, that's true. In most of the tests I have run, there is really very little difference with the previous formatting capability I covered above.   The real benefit here is that f-strings make the instructions easier to read and debug.

Python 3 Notes

I cover three formatting capabilities in Python because you will run into all three. There is actually a fourth capability through the Template class in the old string module. It is very old, and I have never seen it used.

Python 3 Notes

SheBang Line and Encoding Declaration

In computing, a shebang (also called a hashbang, hashpling, pound bang, or crunchbang) refers to the characters "#!" when they are the first two characters in an interpreter directive as the first line of a text file. In a Unix-like operating system, the program loader takes the presence of these two characters as an indication that the file is a script, and tries to execute that script using the interpreter specified by the rest of the first line in the file.

From Python 3 Documentation:

To easily use Python scripts on Unix, you need to make them executable, e.g. with:

```
$ chmod +x script
```

and put an appropriate shebang line at the top of the script. A good choice is usually:

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

which searches for the Python interpreter in the whole `PATH`. However, some Unixes may not have the **env** command, so you may need to hardcode `/usr/bin/python` as the interpreter path. To use shell commands in your Python scripts, look at the `subprocess` module.

In Python 3 the default encoding for your source code is UTF-8. If you are using any other encoding, it must be designated at the top of the program immediately following the shebang if present. More detailed information can be found here. While the minimal requirement for this encoding is different, the standard declaration statement is of the format:

```
# -*- coding: <encoding name> -*-
```

Example:

```
# -*- coding: latin-1 -*-
```

You can avoid this specification if you only use non-ascii characters inside literals. Also, if you are reading files that contain non-ascii characters, the encoding must be specified in the open function. In this case the encoding needn't be specified at the top of your program.

Reserved Words:

The following list shows the reserved words in Python. These reserved words may not be used as constant or variable or any other identifier names.

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	`break`
`except`	`in`	`raise`		

String Methods

Strings implement all of the common sequence operations, along with the additional methods described below:

`str.capitalize`()

Return a copy of the string with its first character capitalized and the rest lowercased.

`str.casefold`()

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, `lower()` would do nothing to `'ß'`; `casefold()` converts it to `"ss"`.

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.center`(*width*[, *fillchar*])

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.count`(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

`str.encode`(*encoding="utf-8", errors="strict"*)

Return an encoded version of the string as a bytes object. Default encoding is `'utf-8'`. *errors* may be given to set a different error handling scheme. The default for *errors* is `'strict'`, meaning that encoding errors raise a `UnicodeError`. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via `codecs.register_error()`, see section Error Handlers. For a list of possible encodings, see section Standard Encodings.

Changed in version 3.1: Support for keyword arguments added.

`str.endswith`(*suffix*[, *start*[, *end*]])

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

`str.expandtabs`(*tabsize=8*)

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed.

>>>

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

`str.find`(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

Note

The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

>>>

```
>>> 'Py' in 'Python'
True
```

`str.format`(**args*, ***kwargs*)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

>>>

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See Format String Syntax for a description of the various formatting options that can be specified in format strings.

`str.format_map`(*mapping*)

Similar to `str.format(**mapping)`, except that `mapping` is used directly and not copied to a `dict`. This is useful if for example `mapping` is a dict subclass:

>>>

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

New in version 3.2.

`str.index`(*sub*[, *start*[, *end*]])

Like `find()`, but raise `ValueError` when the substring is not found.

`str.isalnum`()

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise. A character `c` is alphanumeric if one of the following returns `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

`str.isalpha`()

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise. Alphabetic characters are those characters defined in the Unicode character database as "Letter", i.e., those with general category property being one of "Lm", "Lt", "Lu", "Ll", or "Lo". Note that this is different from the "Alphabetic" property defined in the Unicode Standard.

`str.isdecimal`()

Return true if all characters in the string are decimal characters and there is at least one character, false otherwise. Decimal characters are those from general category "Nd". This category includes digit characters, and all characters that can be used to form decimal-radix numbers, e.g. U+0660, ARABIC-INDIC DIGIT ZERO.

`str.isdigit()`

Return true if all characters in the string are digits and there is at least one character, false otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

`str.isidentifier()`

Return true if the string is a valid identifier according to the language definition, section Identifiers and keywords.

Use `keyword.iskeyword()` to test for reserved identifiers such as `def` and `class`.

`str.islower()`

Return true if all cased characters [4] in the string are lowercase and there is at least one cased character, false otherwise.

`str.isnumeric()`

Return true if all characters in the string are numeric characters, and there is at least one character, false otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

`str.isprintable()`

Return true if all characters in the string are printable or the string is empty, false otherwise. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

`str.isspace()`

Return true if there are only whitespace characters in the string and there is at least one character, false otherwise. Whitespace characters are those characters defined in the Unicode character database as “Other” or “Separator” and those with bidirectional property being one of “WS”, “B”, or “S”.

`str.istitle()`

Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

`str.isupper()`

Return true if all cased characters [4] in the string are uppercase and there is at least one cased character, false otherwise.

`str.join`(*iterable*)

Return a string which is the concatenation of the strings in the iterable *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

`str.ljust`(*width*[, *fillchar*])

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.lower()`

Return a copy of the string with all the cased characters [4] converted to lowercase.

The lowercasing algorithm used is described in section 3.13 of the Unicode Standard.

`str.lstrip`([*chars*])

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

>>>

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

static `str.maketrans`(*x*[, *y*[, *z*]])

This static method returns a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to Unicode ordinals, strings (of arbitrary lengths) or None. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`str.partition`(*sep*)

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

`str.replace`(*old*, *new*[, *count*])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.rfind`(*sub*[, *start*[, *end*]])

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.rindex`(*sub*[, *start*[, *end*]])

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

`str.rjust`(*width*[, *fillchar*])

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.rpartition`(*sep*)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

`str.rsplit`(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

`str.rstrip`([*chars*])

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>>
```

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

`str.split`(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>>
```

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

>>>

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

`str.splitlines`([*keepends*])

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of universal newlines.

Representation	**Description**
`\n`	Line Feed
`\r`	Carriage Return
`\r\n`	Carriage Return + Line Feed
`\v` or `\x0b`	Line Tabulation
`\f` or `\x0c`	Form Feed
`\x1c`	File Separator
`\x1d`	Group Separator
`\x1e`	Record Separator
`\x85`	Next Line (C1 Control Code)
`\u2028`	Line Separator
`\u2029`	Paragraph Separator

Changed in version 3.2: `\v` and `\f` added to list of line boundaries.

For example:

>>>

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike `split()` when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

>>>

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, `split('\n')` gives:

>>>

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.startswith`(*prefix*[, *start*[, *end*]])

Return `True` if string starts with the *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

`str.strip`([*chars*])

Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

>>>

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

The outermost leading and trailing *chars* argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in *chars*. A similar action takes place on the trailing end. For example:

>>>

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.swapcase()`

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

`str.title()`

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

For example:

>>>

```
>>> 'Hello world'.title()
'Hello World'
```

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

>>>

```
>>> "they're bill's friends from the UK".title()
"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

>>>

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

`str.translate(`*table*`)`

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via `__getitem__()`, typically a mapping or sequence. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a `LookupError` exception, to map the character to itself.

You can use `str.maketrans()` to create a translation map from character-to-character mappings in different formats.

See also the `codecs` module for a more flexible approach to custom character mappings.

`str.upper()`

Return a copy of the string with all the cased characters [4] converted to uppercase. Note that `str.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not "Lu" (Letter, uppercase), but e.g. "Lt" (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 of the Unicode Standard.

`str.zfill(`*width*`)`

Return a copy of the string left filled with ASCII `'0'` digits to make a string of length *width*. A leading sign prefix (`'+'`/`'-'`) is handled by inserting the padding *after* the sign character rather than before. The original string is returned if *width* is less than or equal to `len(s)`.

For example:

>>>

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

List Methods

Here are all of the methods of list objects:

`list.append`(*x*) - Add an item to the end of the list.

`list.extend`(*L*) - Extend the list by appending all the items in the given list.

`list.insert`(*i*, *x*) - Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.remove`(*x*) - Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.pop`([*i*]) - Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.clear`() - Remove all items from the list. Equivalent to `del a[:]`.

`list.index`(*x*) - Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

`list.count`(*x*) - Return the number of times *x* appears in the list.

`list.sort`() - Sort the items of the list in place.

`list.reverse`() - Reverse the elements of the list in place.

`list.copy`() - Return a shallow copy of the list. Equivalent to `a[:]`.

A summary of list operations and methods can be found here. These operations apply to any mutable sequence.

Dictionary Methods

Python includes the following dictionary methods:

dict.clear() - Removes all elements of dictionary

dict.copy() - Returns a shallow copy of dictionary

dict.fromkeys(seq, [value]) - Create a new dictionary with keys from seq and values set to value or None.

dict.get(key, default=None) - For key key, returns value or default if key not in dictionary.

dict.has_key(key) - Returns true if key in dictionary, false otherwise

dict.items() - Returns an iterator of dictionary (key, value) tuple pairs

dict.keys() - Returns an iterator of dictionary keys

dict.values() - Returns an iterator of dictionary values

dict.pop(key, [default]) – remove key and return value. Return default if key not found or error if default not provided.

dict.setdefault(key, default=None) - Similar to get(), but will set dict[key]=default if key is not already in dict

dict.update(dict2) - Adds dictionary dict2's key-values pairs to dict. Dict2 values replace any duplicate keys in dict.

File Methods

These are taken from Tutorials Point. Each method below is a link to the Tutorials Point explanation for that method.

Methods with Description

file.close()
Close the file. A closed file cannot be read or written any more.

file.flush()
Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.

file.fileno()
Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.

file.isatty()
Returns True if the file is connected to a tty(-like) device, else False.

file.read([size])
Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).

file.readline([size])
Reads one entire line from the file. A trailing newline character is kept in the string.

file.readlines([sizehint])
Reads until EOF using readline() and returns a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

file.seek(offset[, whence])
Sets the file's current position

file.tell()
Returns the file's current position

file.truncate([size])
Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.

file.write(str)
Writes a string to the file. There is no return value.

file.writelines(sequence)
Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.

Set Methods/Operations

For a set named s:

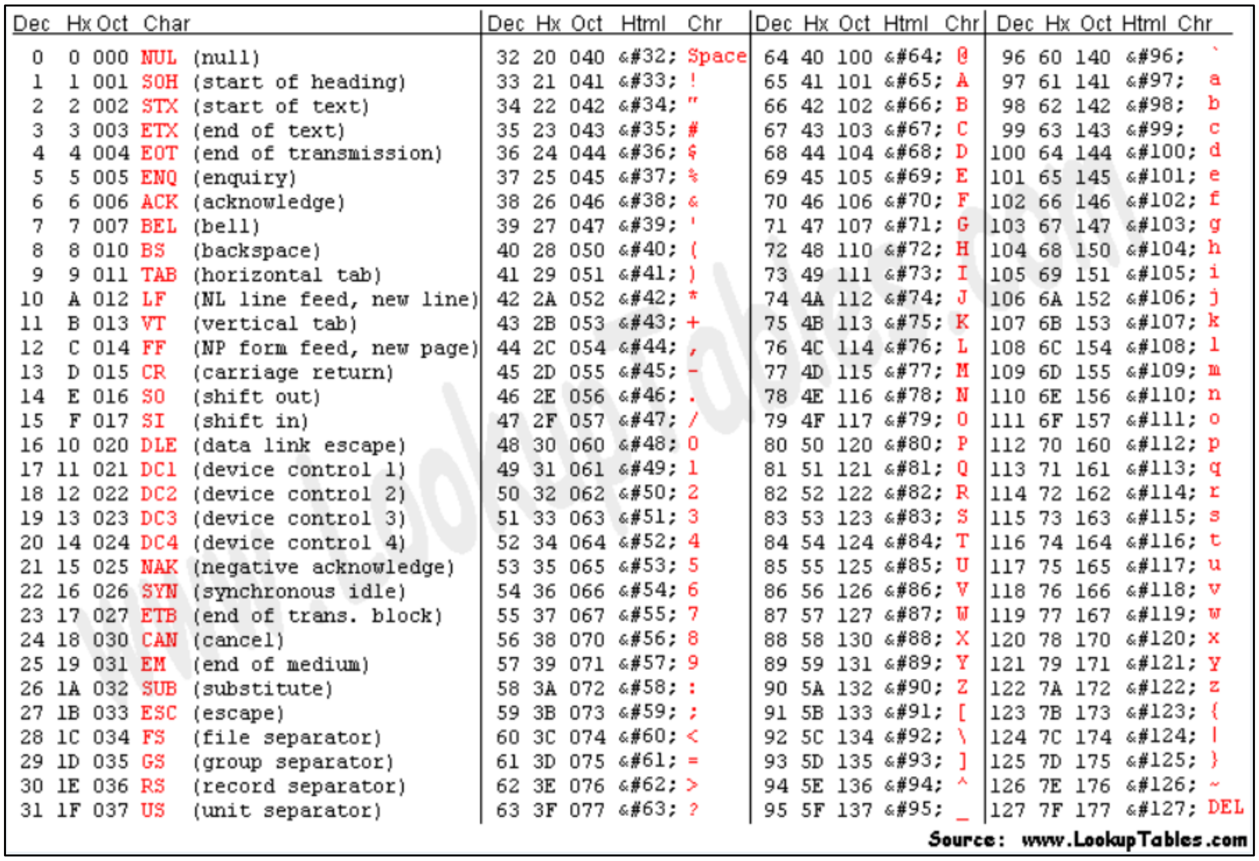
Operation	**Equivalent**	**Result**
len(s)		cardinality of set *s*
x in s		test *x* for membership in *s*
x not in s		test *x* for non-membership in *s*
s.issubset(t)	`s <= t`	test whether every element in *s* is in *t*
s.issuperset(t)	`s >= t`	test whether every element in *t* is in *s*
s.isdisjoint(t)		Return True if the set s has no elements in common with t.
s.union(t)	`s | t`	new set with elements from both *s* and *t*
s.intersection(t)	`s & t`	new set with elements common to *s* and *t*
s.difference(t)	`s - t`	new set with elements in *s* but not in *t*
s.symmetric_difference(t)	`s ^ t`	new set with elements in either *s* or *t* but not both
s.copy()		new set with a shallow copy of *s*

Note – for union, intersection and difference methods, t can be more than one set. For example:
s.difference(t, u, v) or s – t – u – v

Sets are mutable and can be changed in place with these operations.

Operation	**Equivalent**	**Result**
s.update(t)	s |= t	return set s with elements added from t
s.intersection_update(t)	s &= t	return set s keeping only elements also found in t
s.difference_update(t)	s -= t	return set s after removing elements found in t
s.symmetric_difference_update(t)	s ^= t	return set s with elements from s or t but not both
s.add(x)		add element x to set s
s.remove(x)		remove x from set s; raises KeyError if not present
s.discard(x)		removes x from set s if present
s.pop()		remove and return an arbitrary element from s; raises KeyError if empty
s.clear()		remove all elements from set s

ASCII Table



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com