Python Notes

Table of Contents

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 2.7.5 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.

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

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 installations come with a version of Python and most of these are 2.7. Unfortunately, CentOS6 comes with 2.6. CentOS7 comes with 2.7. Most of what we do in Foundations I and II works just fine in 2.6, but 2.7 is much better to work in as it is the more current release. Also, it contains many of the transitional capabilities that allow you to start getting used to version 3 of Python. The Linux repository also contains version 3 of Python. Currently, it is 3.5. With Ubuntu 15.04 and 15.10 you get both 2.7 and 3.4 versions of Python.

All of the above require administrator privileges for installation. An alternative is to use Miniconda. 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 through command line unless you want to dig through the various folders and find IDLE. Just issue idle as a command in terminal or PowerShell and IDLE will appear. Leave that copy of terminal or PowerShell alone until you want to shut down IDLE.

Another option is to use Python Tutor which allows you to code online and execute your program one line at a time or all at once. In the initial window, choose Start visualizing your code now, choose the language you want (Python 2 or 3 – note the other languages you can use), and start entering your code. When you want to test your code, click on Visualize Execution and choose whether you want to execute one instruction at a time (Forward) or the entire program (Last). There doesn't appear to be much in the way of help to use this site other than the limited information you get on the first screen.

A final option is Python Anywhere. You have to sign up for the service and there is a free version. It appears to be much more comprehensive than Python Tutor. Also, it relieves you from having to install Python on your system.

An excellent book for learning Python 2.7 can be found at http://www.pythonlearn.com/book.php. (This book is supplied in class in PDF format.) Just choose the format you want at this link. This book avoids the mathematics found in the following book and concentrates on the mechanics of the Python language. It also goes into some of the more interesting aspects of the language.

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:
Learning Python in general:

- http://www.tutorialspoint.com/python/ (downloadable as a PDF)
- http://www.pythonforbeginners.com/python-overview-start-here/
- http://www.python-course.eu/index.php

Python Notes

- http://www.codecademy.com/tracks/python

The philosophy of Python according to Tim Peters

Using Python 3.X: http://getpython3.com/diveintopython3/

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

Formatting for Printing

Original 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'
```

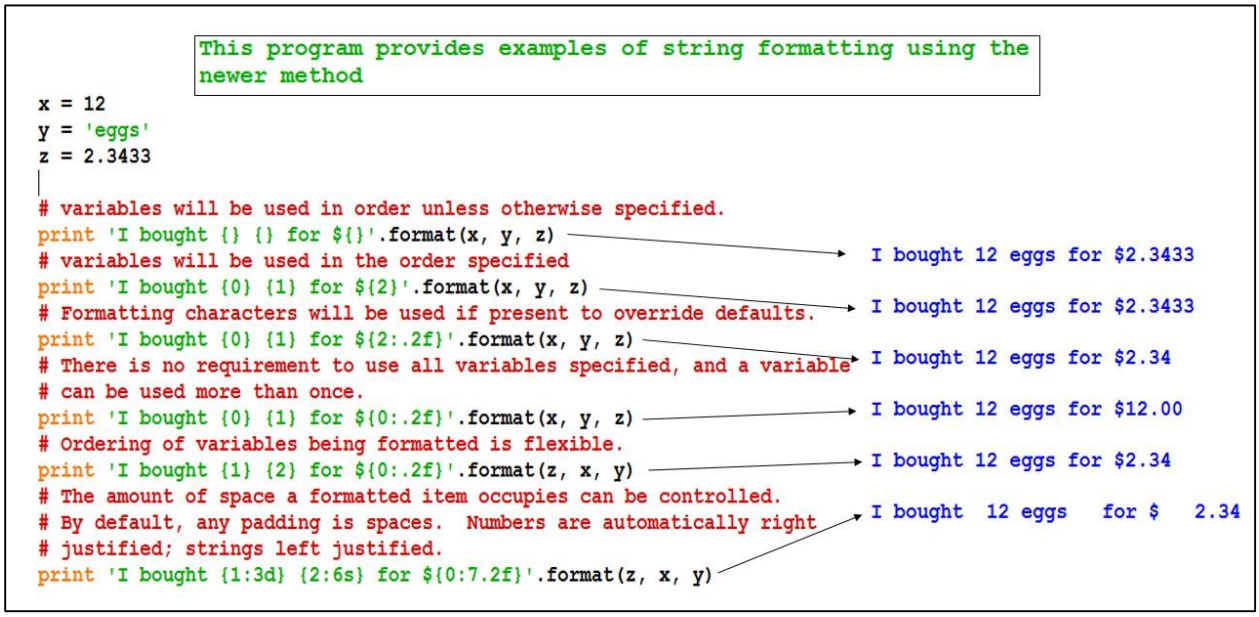
Some more examples of formatting. Note how the floating point number is rounded.

```
>>> 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:

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.
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)

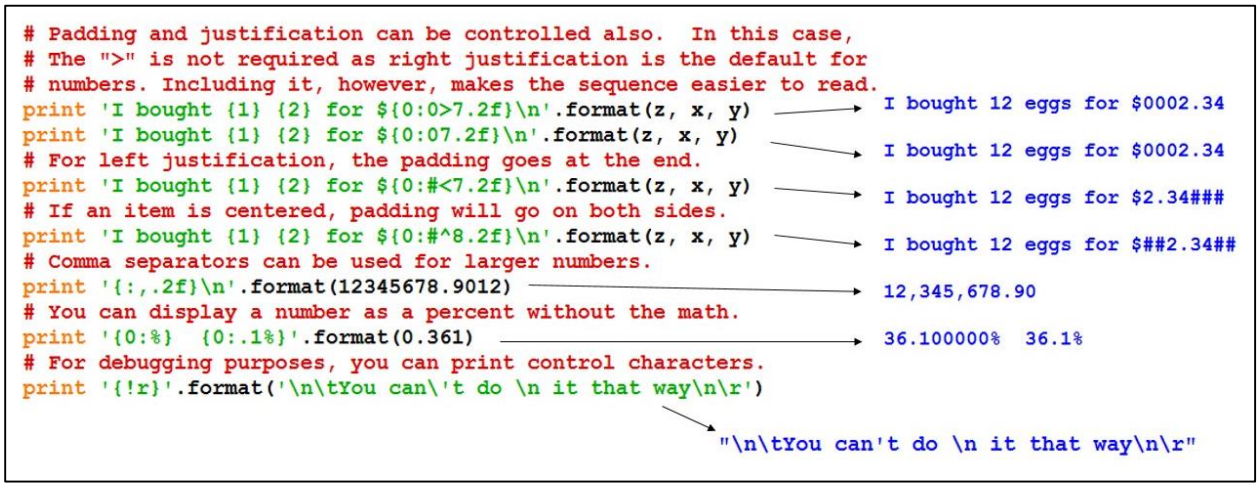
Example snapshots:



```
"""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 ${}'.format(x, y, z)
# variables will be used in the order specified
print 'I bought {0} {1} for ${2}'.format(x, y, z)
# Formatting characters will be used if present to override defaults.
print 'I bought {0} {1} for ${2:.2f}'.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}'.format(x, y, z)
# Ordering of variables being formatted is flexible.
print 'I bought {1} {2} for ${0:.2f}'.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)
```

```
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
```



```
# 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)
# Comma separators can be used for larger numbers.
print '{:,.2f}\n'.format(12345678.9012)
# 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 '{!r}'.format('\n\tYou can\'t do \n it that way\n\r')
```

```
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##
12,345,678.90
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.

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 2 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 2 the default encoding is ASCII. 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 in PEP 263. While the minimal requirement for this encoding is different, the standard declaration statement is of the format:
-*- coding: <encoding name> -*-
Examples:
-*- coding: utf-8 -*- or,
-*- coding: latin-1 -*-

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.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

String Methods

These are the string methods which both 8-bit strings and Unicode objects support:

capitalize()

Return a copy of the string with only its first character capitalized.

For 8-bit strings, this method is locale-dependent.

center(*width*)

Return centered in a string of length *width*. Padding is done using spaces.

count(*sub*[, *start*[, *end*]])

Return the number of occurrences of substring *sub* in string S`[`*start*`:`*end*`]`. Optional arguments *start* and *end* are interpreted as in slice notation.

decode([*encoding*[, *errors*]])

Decodes the string using the codec registered for *encoding*. *encoding* defaults to the default string encoding. *errors* may be given to set a different error handling scheme. The default is `'strict'`, meaning that encoding errors raise `ValueError`. Other possible values are `'ignore'` and `'replace'`. New in version 2.2.

encode([*encoding*[,*errors*]])

Return an encoded version of the string. Default encoding is the current default string encoding. *errors* may be given to set a different error handling scheme. The default for *errors* is `'strict'`, meaning that encoding errors raise a `ValueError`. Other possible values are `'ignore'` and `'replace'`. New in version 2.0.

endswith(*suffix*[, *start*[, *end*]])

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

expandtabs([*tabsize*])

Return a copy of the string where all tab characters are expanded using spaces. If *tabsize* is not given, a tab size of `8` characters is assumed.

find(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start*, *end*). Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

index(*sub*[, *start*[, *end*]])

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

isalnum()

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

isalpha()

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

isdigit()

Return true if all characters in the string are digits and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

islower()

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

For 8-bit strings, this method is locale-dependent.

isspace()

Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

For 8-bit strings, this method is locale-dependent.

istitle()

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

For 8-bit strings, this method is locale-dependent.

isupper()

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

For 8-bit strings, this method is locale-dependent.

join(*seq*)

Return a string which is the concatenation of the strings in the sequence *seq*. The separator between elements is the string providing this method.

ljust(*width*)

Return the string left justified in a string of length *width*. Padding is done using spaces. The original string is returned if *width* is less than `len(s)`.

lower()

Return a copy of the string converted to lowercase.

For 8-bit strings, this method is locale-dependent.

lstrip([*chars*])

Return a copy of the string with leading characters removed. If *chars* is omitted or `None`, whitespace characters are removed. If given and not `None`, *chars* must be a string; the characters in the string will be stripped from the beginning of the string this method is called on. Changed in version 2.2.2: Support for the *chars* argument.

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.

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.

rindex(*sub*[, *start*[, *end*]])

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

rjust(*width*)

Return the string right justified in a string of length *width*. Padding is done using spaces. The original string is returned if *width* is less than `len(s)`.

rstrip([*chars*])

Return a copy of the string with trailing characters removed. If *chars* is omitted or `None`, whitespace characters are removed. If given and not `None`, *chars* must be a string; the characters in the string will be stripped from the end of the string this method is called on. Changed in version 2.2.2: Support for the *chars* argument.

split([*sep* [,*maxsplit*]])

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 is zero, then there is no limit on the number of splits (all possible splits are made). 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 an empty list.

If *sep* is not specified or is `None`, a different splitting algorithm is applied. Words are separated by arbitrary length strings of whitespace characters (spaces, tabs, newlines, returns, and formfeeds). Consecutive whitespace delimiters are treated as a single delimiter ("`'1 2 3'.split()`" returns "`['1', '2', '3']`"). Splitting an empty string returns "`['']`".

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.

startswith(*prefix*[, *start*[, *end*]])

Return `True` if string starts with the *prefix*, otherwise return `False`. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

strip([*chars*])

Return a copy of the string with leading and trailing characters removed. If *chars* is omitted or `None`, whitespace characters are removed. If given and not `None`, *chars* must be a string; the characters in the string will be stripped from the both ends of the string this method is called on. Changed in version 2.2.2: Support for the *chars* argument.

`swapcase`()

Return a copy of the string with uppercase characters converted to lowercase and vice versa.

For 8-bit strings, this method is locale-dependent.

`title`()

Return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

For 8-bit strings, this method is locale-dependent.

`translate`(*table*[, *deletechars*])

Return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

For Unicode objects, the `translate()` method does not accept the optional *deletechars* argument. Instead, it returns a copy of the *s* where all characters have been mapped through the given translation table which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or `None`. Unmapped characters are left untouched. Characters mapped to `None` are deleted. Note, a more flexible approach is to create a custom character mapping codec using the `codecs` module (see `encodings.cp1251` for an example).

`upper`()

Return a copy of the string converted to uppercase.

For 8-bit strings, this method is locale-dependent.

`zfill`(*width*)

Return the numeric string left filled with zeros in a string of length *width*. The original string is returned if *width* is less than `len(s)`. New in version 2.2.2.

List Methods

(from: http://pguides.net/python-tutorial/python-list-methods/)

append(item)

Append an item to the end of the list.

```
l = [1,2,3]
l.append(4)
l
=> [1,2,3,4]
```

count(item)

Count the number of occurrences of the given item in the list. *Note:* This function also works on tuples.

```
l = [1,2,3,1,2,1]
l.count(1)
=> 3
l.count(3)
=> 1
l.count(10)
=> 0
```

extend(other_list)

Extend one list with the contents of `other_list`.

```
l = [1,2]
l.extend([3,4])
l
=> [1,2,3,4]
```

index(val)

Returns the index of the first item in the list whose value matches the given value. `ValueError` if no match is found. *Note:* This function also works on tuples.

```
l = [1,2,2,1,3]
l.index(2)
=> 1
l.index(10)
=> ValueError: list.index(x): x not in list
```

insert(pos, item)

Insert the given item at the specified position. If the position is past the end of the list, insert at the end.

```
l = [1,2,3,4]
l.insert(1,10)
l
=> [1,10,2,3,4]
l.insert(10,"end")
l
=> [1,10,2,3,4,"end"]
```

pop([pos])

Remove and return the element at the specified position. If no position is given, defaults to the last element in the list.

```
l = [1,2,3,4,5]
l.pop()
```

```
3 => 5
4 l
5 => [1,2,3,4]
6 l.pop(0)
7 => 1
```

remove(val)

Remove the first element in the list whose value matches the given value. `ValueError` if no match is found.

```
1 l = [1,2,3,2,1]
2 l.remove(2)
3 l
4 => [1,3,2,1]
5 l.remove(10)
6 => ValueError: list.remove(x): x not in list
```

reverse()

Reverse the order of elements within the list. Changes the list in place instead of returning a modified copy.

```
1 l = [1,2,3,4,5]
2 l.reverse()
3 l
4 => [5,4,3,2,1]
```

sort()

Sorts the list in place ordering elements from smallest to largest.

```
1 l = [10,2,3,10,100,54]
2 l.sort()
3 l
4 => [2, 3, 10, 10, 54, 100]
```

Dictionary Methods

(From: http://www.tutorialspoint.com/python/python_dictionary.htm)

Python includes following dictionary methods

dict.clear()
Removes all elements of dictionary *dict*

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

dict.fromkeys()
Create a new dictionary with keys from seq and values *set* to *value*.

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 *dict*, *false* otherwise

dict.items()
Returns a list of *dict*'s (key, value) tuple pairs

dict.keys()
Returns list of dictionary dict's keys

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*

dict.values()
Returns list of dictionary *dict*'s values

From Python Documentation

dict.iteritems()
Return an iterator over the dictionary's (key, value) pairs.

dict.iterkeys()
Return an iterator over the dictionary's keys.

dict.itervalues()
Return an iterator over the dictionary's values..

File Methods

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

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

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

- file.readlines([sizehint])
Read until EOF using readline() and return 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.write(str)
Write a string to the file. There is no return value.

- file.writelines(sequence)
Write 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.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*

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

Source: www.LookupTables.com