Python CheatSheet (For Linux)
Beginning Python: From Novice to Professional
Chapter 3(Strings)

String Formatting

String formatting uses the (aptly named) string formatting operator, the percent (%) sign. To the left of the %, you place a string (the format string); to the right of it, you place the value you want to format.

```
>>> format = "Hello, %s. %s enough for ya?"
>>> values = ('world', 'Hot')
>>> print format % values
Hello, world. Hot enough for ya?
```

If you use a list or some other sequence instead of a tuple, the sequence will be interpreted as a single value. Only tuples and dictionaries will allow you to format more than one value.

The **%s** parts of the format string are called **conversion specifiers**.

To actually include a percent sign in the format string, you must write **%%** so Python doesn't mistake it for the beginning of a conversion specifier.

If you are formatting real numbers (floats), you can use the f specifier type and supply the precision as a . (dot), followed by the number of decimals you want to keep. The format specifier always ends with a type character, so you must put the precision before that:

```
>>> format = "Pi with three decimals: %.3f"
>>> from math import pi
>>> print format % pi
Pi with three decimals: 3.142
```

Template Strings

The string module offers another way of formatting values: template strings. They work more like variable substitution in many UNIX shells, with $foo being replaced by a keyword argument called foo, which is passed to the template method substitute:

```
>>> from string import Template
>>> s = Template('$x, glorious $x!')
>>> s.substitute(x='slurm')
'slurm, glorious slurm!'
```

If the replacement field is part of a word, the name must be enclosed in **braces**, in order to clearly indicate where it ends:

```
>>> s = Template("It's ${x}tastic!")
>>> s.substitute(x='slurm')
"It's slurmtastic!"
```

In order to insert a dollar sign, use **$$**:

```
>>> s = Template("Make $$ selling $x!")
>>> s.substitute(x='slurm')
'Make $ selling slurm!'
```

Instead of using keyword arguments, you can supply the value-name pairs in a dictionary:

```
>>> s = Template('A $thing must never $action.')
>>> d = {}
>>> d['thing'] = 'gentleman'
>>> d['action'] = 'show his socks'
>>> s.substitute(d)
'A gentleman must never show his socks.'
```

If the right operand is a tuple, each of its elements is formatted separately, and you need a conversion specifier for each of the values.

```
>>> '%s plus %s equals %s' % (1, 1, 2)
'1 plus 1 equals 2'
```

A basic conversion specifier (as opposed to a full conversion specifier, which may contain a mapping key as well) consists of the items that follow. Note that the order of these items is crucial.
• **The % character:** This marks the beginning of the conversion specifier.
• **Conversion flags:** These are optional and may be -, indicating left alignment; +, indicating that a sign should precede the converted value; " " (a space character), indicating that a space should precede positive numbers; or 0, indicating that the conversion should be zero-padded.
• **The minimum field width:** This is also optional and specifies that the converted string will be at least this wide. If this is an * (asterisk), the width will be read from the value tuple.
• **A . (dot) followed by the precision:** This is also optional. If a real number is converted, this many decimals should be shown. If a string is converted, this number is the maximum field width. If this is an * (asterisk), the precision will be read from the value tuple.
• **The conversion type:** This can be any of the types listed below.

Conversion Type	Meaning
`d, i`	Signed integer decimal
`o`	Unsigned octal
`u`	Unsigned decimal
`x`	Unsigned hexadecimal (lowercase)
`X`	Unsigned hexadecimal (uppercase)
`e`	Floating-point exponential format (lowercase)
`E`	Floating-point exponential format (uppercase)
`f, F`	Floating-point decimal format
`g`	Same as `e` if exponent is greater than –4 or less than precision; `f` otherwise
`G`	Same as `E` if exponent is greater than –4 or less than precision; `F` otherwise
`c`	Single character (accepts an integer or a single character string)
`r`	String (converts any Python object using `repr`)
`s`	String (converts any Python object using `str`)

Width and Precision

A conversion specifier may include a field **width and a precision.** The width is the minimum number of characters reserved for a formatted value. The precision is (for a numeric conversion) the number of decimals that will be included in the result or (for a string conversion) the maximum number of characters the formatted value may have. These two parameters are supplied as two integer numbers (width first, then precision), separated by a . (dot). Both are optional, but if you want to supply only the precision, you must also include the dot:

```
>>> '%10f' % pi
' 3.141593'
```

Signs, Alignment, and Zero-Padding

Before the width and precision numbers, you may put a "flag," which may be either zero, plus, minus, or blank.
A zero means that the number will be zero-padded:

```
>>> '%010.2f' % pi
'0000003.14'
```

A minus sign (-) left-aligns the value:

```
>>> '%-10.2f' % pi
'3.14
```

A blank (“ ”) means that a blank should be put in front of positive numbers.

```
>>> print ('% 5d' % 10) + '\n' + ('% 5d' % -10)F
10
-10
```

Finally, a plus (+) means that a sign (either plus or minus) should precede both positive and negative numbers (again, useful for aligning):

```
>>> print ('%+5d' % 10) + '\n' + ('%+5d' % -10)
+10
-10
```

String Methods

Even though string methods have completely upstaged the string module.

string.digits: A string containing the digits 0–9
string.letters: A string containing all letters (uppercase and lowercase)
string.lowercase: A string containing all lowercase letters
string.printable: A string containing all printable characters
string.punctuation: A string containing all punctuation characters
string.uppercase: A string containing all uppercase letters

Note that the string constant letters (such as string.letters) are locale-dependent (that is, their exact values depend on the language for which Python is configured). 3 If you want to make sure you’re using ASCII, you can use the variants with ascii_ in their names, such as string.ascii_letters.

Find

The find method finds a substring within a larger string. It returns the leftmost index where the substring is found. If it is not found, –1 is returned:

```
>>> 'With a moo-moo here, and a moo-moo there'.find('moo')
7
>>> title = "Monty Python's Flying Circus"
>>> title.find('Monty')
0
```

join

A very important string method, join is the inverse of split. It is used to join the elements of a sequence:

```
>>> seq = ['1', '2', '3', '4', '5']
>>> sep.join(seq) # Joining a list of strings
'1+2+3+4+5'
>>> dirs = '', 'usr', 'bin', 'env'
>>> '/'.join(dirs)
'/usr/bin/env'
>>> print 'C:' + '\\'.join(dirs)
C:\usr\bin\env
```

lower

The lower method returns a lowercase version of the string:

```
>>> 'Trondheim Hammer Dance'.lower()
'trondheim hammer dance'
```

replace

The replace method returns a string where all the occurrences of one string have been replaced by another:

```
>>> 'This is a test'.replace('is', 'eez')
'Theez eez a test'
```

If you have ever used the “**search and replace**” feature of a word processing program, you will no doubt see the usefulness of this method.

split

A very important string method, split is the inverse of join, and is used to split a string into a sequence:

```
>>> '1+2+3+4+5'.split('+')
['1', '2', '3', '4', '5']
>>> '/usr/bin/env'.split('/')
['', 'usr', 'bin', 'env']
>>> 'Using the
default'.split()
['Using', 'the', 'default']
```

strip

The strip method returns a string where whitespace on the left and right (but not internally) has been stripped (removed):

```
>>> '     internal whitespace is kept    '.strip()
'internal whitespace is kept'
```

translate

Similar to replace, translate replaces parts of a string, but unlike replace, translate works only with single characters. Its strength lies in that it can perform several replacements simultaneously, and can do so more efficiently than replace.

```
>>> from string import maketrans
>>> table = maketrans('cs', 'kz')
>>> 'this is an incredible test'.translate(table)
'thiz iz an inkredible tezt'
```

Python doesn't consider Ø a real letter.

New Functions in This Chapter

Function	Description
`string.capwords(s[, sep])`	Splits `s` with `split` (using `sep`), capitalize items, and join with a single space
`string.maketrans(from, to)`	Makes a translation table for `translate`