DD1311 Programmeringsteknik med PBL
Per Sedholm
13 mars 2008

Formaterade utskrifter

Väldigt många program behöver hantera text, om inte annat så för att visa upp något resultat för användaren.

I de flesta fall vill man inte bara vill skriva ut någonting som text, utan som *formaterad* text. Exempelvis, skriva ut siffror med ett visst antal decimaler, eller skriva ut en snygg tabell med värden.

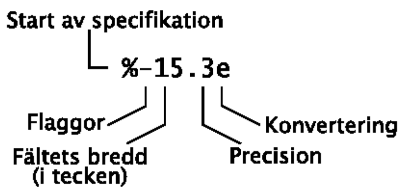
De flesta programmeringsspråk har någon variant av `printf` (*print formatted*, från språket C:s funktion med samma namn). Jämför till exempel

Språk	Exempel
C	`printf("pi = %.2f\n", 3.1415926)`
Matlab	`disp(sprintf('pi = %.2f', 3.1415926))`
Perl	`printf("pi = %.2f\n", 3.1415926)`
Python	`print "pi = %.2f" % 3.1415926`
Java	`System.out.printf("pi = %.2f\n", 3.1415926)`

Alla fungerar på ungefär samma sätt: man ger en *formatmall*, och ett eller flera värden som ska formateras enligt den. Resultatet – en följd av tecken – skrivs sedan ut, eller returneras i form av en sträng (som sedan kan skrivas ut).

I Python används %-operatorn för att formatera (eller interpolera) värden som strängar[1]. Man anropar den som ***format* % *värden***, där *format* är en sträng, och *värden* är ett eller flera värden. Är det flera värden används en tupel, lista eller annan sekvensiell datatyp som högerled.

Formatmallen innehållar vanligtvis *formatspecifikationer*, eller *fält*, och dessa ska passa typen på de parametrar man skickar (heltal/flyttal/sträng/etc.). Ifall man sagt att något ska formateras som ett flyttal med tre decimaler, så kan man inte skicka värdet `"röd"`.



Det enda av elementen som krävs är det som specificerar konvertering. Övriga är valbara.

Flaggor

Flagga	Betydelse
`-` (minustecken)	Vänsterställ inom fältet
`+` (plustecken)	Skriv alltid tecken (+ eller −)
`0` (nolla)	Fyll ut med nollor istället för mellanslag

Bredd, precision

Fältets *bredd* är antalet tecken som skall användas, inklusive såna som går åt till att skriva minustecken ("−"), decimalpunkt (".") etc. Efter att fältet formaterats, fylls det ut med mellanslag ("␣") eller nollor ("0").

```
>>> pi = 3.14159265358979323846
>>> print "pi ::%10.3f::" % pi
pi ::     3.142::
>>> print "pi ::%010.3f::" % pi
pi ::000003.142::
```

[1] Operatorn % används också för modulo-räkning (rest vid heltalsdivision) ifall man använder den på siffror.

Precisionen är antalet siffror efter decimalpunkten,[2] eller (för strängar) maximala antalet tecken som ska tas från strängen.

```
>>> print "::%5s::" % "hej"
::  hej::
>>> print "::%5s::" % "hello world"
::hello world::
>>> print "::%5.5s::" % "hello world"
::hello::
```

Konvertering

Följande är de vanligaste konverteringarna. De förekommer i de flesta programmeringsspråk. Fler finns, se dokumentationen för detaljer.

Tecken	Argumenttyp; Konvertering
`d, i`	Heltal, skrivet decimalt (talbas 10) med minustecken (−) för negativa tal.
`x, o`	Heltal, skrivet hexadecimalt (talbas 16) eller oktalt (talbas 8).
`f`	Flyttal, skrivet på formen `[-]d.ddd`. Om inte annat anges så används 6 tecken efter decimalpunkten.
`e`	Flyttal skrivet som `[-]d.ddde`±`dd`, dvs `3.142e+01` för att skriva $3.142 \cdot 10^{01}$ (dvs 31.42).
`g`	Flyttal, som `f` eller `e` beroende på talet som konverteras.
`s`	Sträng.
`%`	%-tecken; inget argument att konvertera

Exempel

```
names = { 'Adam' : 23,
          'Stina' : 18,
          'Göran' : 24 }
for name, age in names.items():
    print "%-10s %d" % (name, age)
```

```
Stina      18
Göran      24
Adam       23
```

```
class Parrot(object):
    def __init__(self, species, age, health):
        self.species = species
        self.age = age
        self.health = health

parrots = [Parrot('Arctic Grey', 8, 'sleeping'),
           Parrot('Norwegian Blue', 12, 'dead'),
           Parrot('Cockatiel', 20, 'an ex-parrot')]
for p in parrots:
    print "This %s is %d years old; it's %s"\
          % (p.species, p.age, p.health)
```

```
This Arctic Grey is 8 years old; it's sleeping
This Norwegian Blue is 12 years old; it's dead
This Cockatiel is 20 years old; it's an ex-parrot
```

[2] Använd “0” för att inte få någon decimalpunkt.

5.6 Binary arithmetic operations

Excerpt from *The Python Reference Manual*
http://docs.python.org/ref/binary.html

In addition to performing the modulo operation on numbers, the % operator is also overloaded by string and unicode objects to perform string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section "Sequence Types".

3.6.2 String Formatting Operations

Excerpt from *The Python Library Reference*
http://docs.python.org/lib/typesseq-strings.html

String and Unicode objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given ***format* % *values*** (where *format* is a string or Unicode object), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to the using `sprintf()` in the C language. If *format* is a Unicode object, or if any of the objects being converted using the `%s` conversion are Unicode objects, the result will also be a Unicode object.

If *format* requires a single argument, *values* may be a single non-tuple object.[3] Otherwise, *values* must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The "`%`" character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an "`*`" (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a "`.`" (dot) followed by the precision. If specified as "`*`" (an asterisk), the actual width is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string *must* include a parenthesised mapping key into that dictionary inserted immediately after the "`%`" character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print '\%(language)s has \%(#)03d quote types.' \% \
         {'language': "Python", "#": 2}
Python has 002 quote types.
```

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
`#`	The value conversion will use the "alternate form" (where defined below).
`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).

A length modifier (`h`, `l`, or `L`) may be present, but is ignored as it is not necessary for Python.

[3]To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

The conversion types are:

Conv.	Meaning	Notes
`d`	Signed integer decimal.	
`i`	Signed integer decimal.	
`o`	Unsigned octal.	(1)
`u`	Unsigned decimal.	
`x`	Unsigned hexadecimal (lowercase).	(2)
`X`	Unsigned hexadecimal (uppercase).	(2)
`e`	Floating point exponential format (lowercase).	(3)
`E`	Floating point exponential format (uppercase).	(3)
`f`	Floating point decimal format.	(3)
`F`	Floating point decimal format.	(3)
`g`	Floating point format. Uses exponential format if exponent is greater than -4 or less than precision, decimal format otherwise.	(4)
`G`	Floating point format. Uses exponential format if exponent is greater than -4 or less than precision, decimal format otherwise.	(4)
`c`	Single character (accepts integer or single character string).	
`r`	String (converts any python object using `repr()`).	(5)
`s`	String (converts any python object using `str()`).	(6)
`%`	No argument is converted, results in a "`%`" character in the result.	

Notes:

1. The alternate form causes a leading zero ("`0`") to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.
2. The alternate form causes a leading `'0x'` or `'0X'` (depending on whether the "`x`" or "`X`" format was used) to be inserted between left-hand padding and the formatting of the number if the leading character of the result is not already a zero.
3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.
4. The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

   The precision determines the number of significant digits before and after the decimal point and defaults to 6.
5. The `%r` conversion was added in Python 2.0. The precision determines the maximal number of characters used.
6. If the object or format provided is a `unicode` string, the resulting string will also be `unicode`.

   The precision determines the maximal number of characters used.

Since Python strings have an explicit length, `%s` conversions do not assume that `'\0'` is the end of the string.

For safety reasons, floating point precisions are clipped to 50; `%f` conversions for numbers whose absolute value is over 1e25 are replaced by `%g` conversions.[4] All other errors raise exceptions.

Additional string operations are defined in standard modules `string` and `re`.

[4]These numbers are fairly arbitrary. They are intended to avoid printing endless strings of meaningless digits without hampering correct use and without having to know the exact precision of floating point values on a particular machine.