str.format()

-or-

How

`"%s, %s" % ("Hello", "world")`

became

`"{}, {}".format("Hello", "world")`

Eric V. Smith
True Blade Systems, Inc.
eric@trueblade.com

- Overview: What and Why?
- Simple Usage
- Format Specification for Basic Types
- Formatting for Your Own Types
- Defining Your Own Templating Language
- Tips and Tricks

Getting our feet wet

- `"My {0} is {1}".format("name", "Eric") -> "My name is Eric"`
- `"{1} is my {0}".format("name", "Eric") -> "Eric is my name"`
- `"My {attr} is {value}".format (attr="name", value="Eric") -> "My name is Eric"`
- `"My {attr} is {0}".format("Eric", attr="name") -> "My name is Eric"`

Friday, February 19, 2010

Just focus on named and positional args.

What str.format() brings

- New string method: **`str.format`** (and in 2.x, **`unicode`**, too).
- New method on all objects (objects format themselves!): **`__format__(self, fmt)`**.
- New built-in: **`format(obj, fmt=None)`**.
- New class: **`string.Formatter`**.

Friday, February 19, 2010

str.format is similar in concept to % formatting for strings.
__format__ is similar in concept to a parameterized __str__ method.
Default object.__format__(self, fmt) just calls format(str(self), fmt).
__format__ is where the extensibility comes in.
__format__ is on all objects in the sense that __len__ is available on all objects.
The relationship between format and __format__ is like len and __len__.

str.format()

- Described in PEP 3101.
- A way to format strings, similar to and in addition to %-formatting and string.Template.
- Uses **{ }** embedded in strings to expand variables.
- First appeared in CPython 2.6 and 3.0. Supported by Jython and IronPython.
- Minor (but important!) improvements made in 2.7 and 3.1.

Friday, February 19, 2010

I'll refer to this technology as "str.format", but it's really a number of related things. By the time we're done, you'll understand how it all relates together.

Isn't %-formatting good enough?

- The primary issue is that it's a binary operator and difficult to enhance or extend. Unlike most other things in Python, it's not a "normal" function with parameters.
- It's not usable with user-defined types. It has zero extensibility hooks.
- It has the wrong precedence. In particular it binds more tightly than **+**:

```
"this is %s" + "a %s" % ("not",
"test")
```


Friday, February 19, 2010

No named parameters, args, kwargs. First argument is the string, second argument is a tuple, a dict, or an object.

The types known to %-formatting are built in and cannot be extended. If you want to format a date, a Decimal, etc., you need to provide some alternate mechanism.

My biggest problem

- Problem with multiple-element tuples.
  `print("result: %s" % result)`
  What happens when result is `(0, 1)`?

Ouch.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: not all arguments
converted during string formatting
```


Friday, February 19, 2010

This is a real problem. I've seen this problem even in the standard library.
Last Tuesday this caused a production system I work on to crash (bad test coverage, admittedly).

Solution

- To protect yourself against unknown parameters, you must always say:
  `print("result: %s" % (result,))`
- How many of us always do that?

More problems with %-formatting

- You can use either named arguments or positional arguments, but not a mixture.
- You must use named arguments for l10n because that's the only way to swap the order of parameters.
- Syntax for named arguments is clunky:
  `"result: %(value)10.10s" % mydict`
- Can't mix named arguments with '**`*`**'.

Friday, February 19, 2010

More problems with %-formatting.
I always forget the trailing 's'. This syntax confuses many people, who rarely see it.

What about string.Template?

- Described in PEP 292.
- Uses `$` (with optional braces) for expansion variables.
- Not really in the same problem space.

More examples

- `"pi={0:.5}".format(math.pi) -> 'pi=3.1416'`
- `"pi={0:.5} or {0:.2}".format (math.pi) -> 'pi=3.1416 or 3.1'`
- `"pi={0.pi} e={0.e}".format(math) -> 'pi=3.141592653359 e=2.71828182846'`

1: After a colon, type-specific formatters can be supplied.
2: With %-formatting you can use the same value multiple times, but only if you used named arguments.
3: '.' is just attribute access. No method calls, nothing fancy. But since evaluating an attribute can run arbitrary code, there's no real protection here.

__getitem__ access

- `"{0[0]}.{0[1]}".format(sys.version_info)` -> `'3.1'`
- `"The {0[thing]}'s due in {0[when]} days".format({'when':3, 'thing': 'homework'})` -> `'The homework's due in 3 days'`
- `"{0[0]}.{0.minor}".format(sys.version_info)` -> `'2.7'`

1: [] used for getitem access (list indexes or dictionary lookups)
2: Here used for dictionary lookups. Notice the keys provided are strings. The rule is: if it “looks like” an integer, convert it to an integer, otherwise use it as a string.
3: Only works in 2.7 or 3.2 (namedtuple for sys.version)

Yet more examples

- `"pi={0.pi:.{n}}".format(math, n=7)`
  `-> 'pi=3.141593'`
- `"i={0:d} {0:X} {0:#b}".format(300)`
  `-> 'i=300 12C 0b100101100'`
- `"{0:*^20}".format("Python") ->`
  `'*******Python*******'`

1: Shows recursive expansion. Only goes one level deep.
2: Shows format specifiers; also shows using the same attribute multiple times.
3. More format specifiers: centered with '*' for a padding char.

Still more examples

- `"{0:%Y-%m-%d}".format(datetime.now())` -> `'2010-02-17'`
- `"My {0} is {2}".format("last name", "Eric", "Smith")` -> `'My last name is Smith'`

Friday, February 19, 2010

1: Shows that types can specify their own specification language. Here, it's strftime.
2: Not all values must be used. We talked about adding support for making this an optional error, but couldn't come up with a good syntax.

I promise, the last example

- It's easy to create a formatting function.
- ```
  f = "{0} is {1:.12f}".format
  f('pi', math.pi) ->
  'pi is 3.141592653590'
  ```


Friday, February 19, 2010

Since it's just a regular function with regular parameters, you can do the regular stuff you'd expect in Python.
  ```

Type conversions

- They start with "**!**" and must come before the format specifier (if any).
- Valid conversions are:
  - **!s** : convert to string using **str()**.
  - **!r** : convert to string using **repr()**.
  - **!a** : convert to ascii using **ascii()** 3.x only.

Type conversions

- `"{0!r}".format(now) -> 'datetime.date(2010, 2, 17)'`
- `"{0!s}".format(now) -> '2010-02-17'`
- `"{0:%Y} {0!s} {0!r}".format(now) -> '2010 2010-02-17 datetime.date (2010, 2, 17)'`
- `"{0!s:#>20}".format(now) -> '##########2010-02-17'`

Assume that now is a datetime.date(2010, 2, 17)
3: Shows advantage over calling repr(): you can decide in the string, not in the code.
4: Shows that once !r, !s, or !a is used, the result is a string, so the format specifier must use the string format specification language.

Improvements in 2.7 and 3.1

- Comma formatting for numeric types:
  `format(1234567, ',')` -> `'1,234,567'`

- If you want numbered, in order replacement values, you can omit the numbers. This is a huge usability improvement!
  `'I have {:#x} {}'.format(16, 'dogs')` -> `'I have 0x10 dogs'`

- **`complex`** is better supported.

Comma formatting works for float, int (long), complex, Decimal.
Comma formatting is not locale aware. It's hard coded for commas and 3 digit grouping (like it's hard coded for dots, not locale-aware decimals).

str.format() vs. format vs. obj.__format__()

- **`format()`** built-in and **`obj.__format__()`** are the building blocks.

- **`str.format()`** parses strings, separates out the **`{}`** parts, does any lookups or conversions, and calls **`format()`** with the calculated object and the supplied format string. It then knits the result back together into its return parameter. This is similar to %-formatting.

Friday, February 19, 2010

1: Objects format themselves! It's fundamental to the concept.
2: Note that it's str.format() that has the curly braces, and the objects themselves that interpret the format specifications. This is key to the whole thing.

object.__format__

- The default implementation is (basically):

```
def __format__(self, fmt):
  return format(str(self), fmt)
```

- DO NOT RELY ON THIS BEHAVIOR!

- 2.6: `format(1+1j,'*^8s')` -> `'*(1+1j)*'`

- 2.7: `format(1+1j,'*^8s')` ->
  `ValueError: Unknown format code 's' for object of type 'complex'`

If you take only one thing from this talk, this should be it.
By relying on this behavior for built-in types (or types you didn't write), you expose yourself to future breakage when/if those types implement their own format specifications. If you rely on it for your own types, you're forced to either break things or implement str's formatting specifications as a subset of your own spec.

What to do?

- If you really want this behavior, convert to a string first.
- `format(str(1+1j), '*^8s')` returns the same thing in 2.6, 2.7, 3.1, 3.2.
- This is equivalent to:

```
'{0!s:*^8}'.format(1+1j)
```

Types implementing __format__

- **`object`**
- **`str`** (and **`unicode`** in 2.x)
- **`int`** (and **`long`** in 2.x)
- **`float`**
- **`complex`**
- **`decimal.Decimal`**
- **`datetime.date`, `.datetime`, `.time`**

Friday, February 19, 2010

object calls format(str(self), fmt)

datetime types use strftime. Others are documented on next slides.

str & unicode

- Very similar to %-formatting.

  ```
  [[fill]align][minimumwidth]
  [.precision][type]
  ```

- Addition of '^' for center alignment.

Numeric types

- Again, similar to %-formatting.

```
[[fill]align][sign][#][0]
[minimumwidth][.precision][type]
```

- New features: '**^**', '**%**', '**b**', '**n**', '' (empty)

Friday, February 19, 2010

^ centered
% as a percentage
b binary
n locale aware formatting
empty like %g, but always with a digit past the decimal

Formatting your own types

- Just implement **`__format__(self, spec)`**.
- Parse the spec however you want. It's your own type-specific language.
- Or, do what Decimal does and treat spec like the built-in **`float`** specification language.
- You'll automatically be useable via the mechanisms I've shown: **`str.format()`** and **`format()`**.

str.format() weaknesses

- Slower than %-formatting.
- Some people dislike the syntax.
- In 2.6 and 3.0, you must always explicitly identify all replacement variables (by name or number).

Friday, February 19, 2010

Some people disike the syntax: There's more software yet to be written in Python than has already been written.

str.format() strengths

- Types that can be formatted are not limited to a few built-in ones.
- Types can format themselves.
- The formatting language can by type-specific.
- In 2.7 and 3.2, numbers can easily have commas.

Friday, February 19, 2010

This is all in addition to the reasons in the PEP 3101 rationale that I mentioned earlier.
1: The few built-in types for %-formatting are the important ones, though!
We don't need a one-size fits all language, like %-formatting. For example, datetime.
With %-formatting it's hard to expose date formatting as part of l10n.

Your own template language

- **`string.Formatter`**: little known, but powerful.
- It's reasonably fast. The important parts are implemented in C (for CPython).
- So, say we want to use vertical bars "|" instead of curly braces. Let's write a custom class.

Friday, February 19, 2010

But first, let's see how string.Template actually works before we customize it

How string.Template works out of the box

```
>>> fmtr = string.Formatter()
>>> fmtr.format('-{0:^10}-', 'abc')
'-    abc     -'

>>> fmt = string.Formatter().format
>>> fmt('-{0:^10}-', 'abc')
'-    abc     -'
```

Custom template class

```
class BarFormatter(string.Formatter):
  def parse(self, template):
    for s, fld in grouper(2,
                          template.split('|')):
      if fld:
        name, _, spec = \
            fld.partition(':')
        yield s, name, spec, None
      else:
        yield s, None, None, None
```


Friday, February 19, 2010

This example isn't meant to be particularly useful but to demonstrate how much can be down with a little code.
grouper() comes from the itertools recipes.
The Python documentation (section 8.1). The tuples are (literal_text, field_name, format_spec, conversion)
You can also override attribute lookup, whether all parameters are consumed, how the conversion specifiers work, etc.
It's very powerful. You can see how it's implemented in string.py in the stdlib.

Using our custom template language

```
>>> fmt = BarFormatter().format
>>> fmt('-|0:^10s|-', 'abc')
'-    abc     -'

>>> f = lambda k, v: \
    fmt('|0:s| is |1:.13f|', k, v)
>>> f('e', math.e)
'e is 2.7182818284590'
```


Friday, February 19, 2010

Look ma, no curly braces! With just 8 lines of code.
This is not a useful example, of course. But it does show what you can do with a little code.

Tips and Tricks

- Migrating a library from %-formatting to **`str.format()`**.
- Delaying instantiation of parameters.

Migrating from %-formatting to str.format()

- Problem: You have a library that exposes a %-formatting interface, you want to migrate to a more expressive **`str.format()`** interface.

- Solution: You support both for a few releases, then eventually only support **`str.format()`**.

Friday, February 19, 2010

There was some work going on to convert %-format strings to str.format strings.
This won't help you when writing a library where you want to change APIs. It will help you move code that contains %-formatting so that it uses str.format, but that's not our problem here.
That conversion is difficult to do correctly for 100% of the cases.

Existing Library

```
class Logger:
    def __init__(self):
        self.fmt = '%(msg)s'
    def log(self, msg):
        args = {'now': datetime.now(),
                'msg': msg}
        s = self.fmt % args
        print(s)
```

Interim Solution

```
class Logger:
    def __init__(self):
        self.fmt = '{msg!s}'
    def log(self, msg):
        args = {'now': datetime.now(),
                'msg': msg}
s = expand_str_mapping(
            self.fmt, args)
# s = self.fmt.format(args)
print(s)
```


Friday, February 19, 2010

Really the same, except for expand_str_mapping instead of %-formatting with a mapping.
The trick, of course, is writing expand_str_mapping.
It's somewhat easier because, as a library writer, you'll know what type of object you'll be passing in. Almost always a mapping, but you could write expand_str_positional instead.
The final version of this will just use s = self.fmt.format(**args)

expand_str_mapping

- Some amount of guessing involved based on the format string, but really only for pathological cases.
- If the format string has a '`%(`' but not a '`{`', use %-formatting.
- If it has a '`{`' but no '`%(`', use **`str.format()`**.
- And if has neither, no expansion needed (or, it doesn't matter which you use).

Friday, February 19, 2010

This is an active project of mine.

The hard part

- What if a format string has both '`{`' and '`%(`'?
- We'll need to parse the string, but even that isn't enough for a format string like:
  `"{abc:%(abc)s}"`
  But I think it can be made good enough.
- This is an ongoing project of mine. I want to convert argparse before it makes it into the standard library. Send me mail if you're interested or have ideas.

Delayed Instantiation

- Problem: Some objects or calculations are expensive, and you don't want to compute them unless they're used.
- But, if you don't control the format string, you might not know if they've being used.
- Solution: Don't instantiate them until they're actually needed.

Delayed Proxy

```
class Delayed:
    __sntnl = object()
    def __init__(self, fn):
        self.__fn = fn
        self.__obj = self.__sntnl
    def __getattr__(self, attr):
        if self.__obj is self.__sntnl:
            self.__obj = self.__fn()
        return getattr(self.__obj,
                       attr)
```

Using Delayed Instantiation

```
class Logger:
    def __init__(self):
        self.fmt = '{msg!s}'

    def log(self, msg):
        print(self.fmt.format(
          now=Delayed(datetime.now),
          moon=Delayed(moon_phase),
          msg=msg))
```

Phase of the moon

- Basic algorithm from: **http://www.daniweb.com/code/post968407.html**
- **def moon_phase(date=None):**
  - Returns a **collections.namedtuple** of: **(status, light)**.

Friday, February 19, 2010

if date is None, use the current date
status is a string describing the phase of the moon

```
>>> logger = Logger()
>>> logger.log('text')
text

>>> logger.fmt = 'phase {moon[0]!r}:
{msg!s}'
>>> logger.log('text')
phase 'waxing crescent (increasing
to full)': text
```

```
>>> logger.fmt = 'phase {moon[0]!r}
({moon.light:.1%}): {msg!s}'
>>> logger.log('text')
phase 'waxing crescent (increasing
to full)' (34.0%): text

>>> logger.fmt = '{now:%Y-%m-%d}:
{msg!r:.10}'
>>> logger.log(sys)
2010-02-19: <module 's

>>> logger.log(3)
2010-02-19: 3
```


Friday, February 19, 2010

Phase of the moon is delay calculated, can call __getitem__ or use attribute access on it to trigger the instantiation
Same for putting the current time into the result

Questions?

eric@trueblade.com

http://trueblade.com/pycon2010