CS 2316 Data Manipulation for Engineers

Text Processing

Christopher Simpkins
chris.simpkins@gatech.edu

String Interpolation with `%`

The old-style (2.X) string format operator, `%`, takes a string with format specifiers on the left, and a single value or tuple of values on the right, and substitutes the values into the string according to the conversion rules in the format specifiers. For example:

```
>>> "%d %s %s %s %f" % (6, ’Easy’, ’Pieces’, ’of’, 3.14)
’6 Easy Pieces of 3.140000’
```

Here are the conversion rules:

- `%s` string
- `%d` decimal integer
- `%x` hex integer
- `%o` octal integer
- `%f` decimal float
- `%e` exponential float
- `%g` decimal or exponential float
- `%%` a literal

String Formatting with %

Specify field widths with a number between % and conversion rule:

```
>>> sunbowl2012 = [(’Georgia Tech’, 21), (’USC’, 7)]
>>> for team in sunbowl2012:
...     print(’%14s %2d’ % team)
...
  Georgia Tech 21
           USC  7
```

Fields right-aligned by default. Left-align with – in front of field width:

```
>>> for team in sunbowl2012:
...     print(’%-14s %2d’ % team)
...
Georgia Tech   21
USC             7
```

Specify *n* significant digits for floats with a `.n` after the field width:

```
>>> ’%5.2f’ % math.pi
’ 3.14’
```

Notice that the field width indludes the decimal point and output is left-padded with spaces.

String Interpolation with `format()`

New-style (3.X) interpolation is done with the string method `format`:

```
>>> "{} {} {} {} {}".format(6, ’Easy’, ’Pieces’, ’of’, 3.14)
’6 Easy Pieces of 3.14’
```

Old-style formats only resolve arguments by position. New-style formats can take values from any position by putting the position number in the `{}` (Notice that positions start with 0):

```
>>> "{4} {3} {2} {1} {0}".format(6, ’Easy’, ’Pieces’, ’of’, 3.14)
’3.14 of Pieces Easy 6’
```

Can also use named arguments, like functions:

```
>>> "{count} pieces of {kind} pie".format(kind=’punkin’, count=3)
’3 pieces of punkin pie’
```

Or dictionaries (note that there’s one dict argument, number 0):

```
>>> "{0[count]} pieces of {0[kind]} pie".format({’kind’:’punkin’,
    ’count’:3})
’3 pieces of punkin pie’
```

String Formatting with `format()`

Conversion types appear after a colon:

```
>>> "{:d} {} {} {} {:f}".format(6, ’Easy’, ’Pieces’, ’of’, 3.14)
’6 Easy Pieces of 3.140000’
```

Argument names can appear before the `:`, and field formatters appear between the `:` and the conversion specifier (note the < and > for left and right alignment):

```
>>> for team in sunbowl2012:
...     print(’{:<14s} {:>2d}’.format(team[0], team[1]))
...
Georgia Tech   21
USC             7
```

You can also unpack the tuple to supply its elements as individual arguments to `format` (or any function) by prepending tuple with $*$:

```
>>> for team in sunbowl2012:
...     print(’{:<14s} {:>2d}’.format(*team))
...
Georgia Tech   21
USC             7
```

String Methods (1 of 4)

We’ve already covered string methods, but they bear reviewing:

- `str.find(`*substr*`)` returns the index of the first occurence of *substr* in `str`

```
>>> 'foobar'.find('o')
1
```

- `str.replace(`*old*`,` *new*`)` returns a copy of `str` with all occurrences of *old* replaced with *new*

```
>>> 'foobar'.replace('bar', 'fighter')
'foofighter'
```

- `str.split(`*delimiter*`)` returns a list of substrings from `str` delimited by *delimiter*

```
>>> 'foobar'.split('ob')
['fo', 'ar']
```

String Methods (2 of 4)

- `str.join(`*iterable*`)` returns a string that is the concatenation of all the elements of *iterable* with `str` in in between each element

```
>>> 'ob'.join(['fo', 'ar'])
'foobar'
```

- `str.strip()` returns a copy of `str` with leading and trailing whitespace removed

```
>>> '  landing   '.strip()
'landing'
```

- `str.rstrip()` returns a copy of `str` with only trailing whitespace removed

```
>>> '  landing   '.rstrip()
'  landing'
```

String Methods (3 of 4)

- `str.rjust(`*width*`)` returns a copy of `str` that is *width* characters or `len(str)` in length, whichever is greater, padded with leading spaces as necessary

```
>>> 'rewards'.rjust(20)
'             rewards'
```

- `str.upper()` returns a copy of `str` with each character converted to upper case.

```
>>> 'CamelCase'.upper()
'CAMELCASE'
```

- `str.isupper()` returns `True` if `str` is all upper case

```
>>> 'CamelCase'.isupper()
False
>>> 'CAMELCASE'.isupper()
True
```

String Methods (4 of 4)

- `str.isdigit()` returns `True` if `str` is all digits

```
>>> '42'.isdigit()
True
>>> '99 bottles of beer'.isdigit()
False
```

- `str.startswith(`*substr-or-tuple*`)` returns `True` if `str` starts with *substr-or-tuple*

```
>>> 'a bang! a whimper'.startswith('a bang')
True
```

- `str.endswith(`*substr-or-tuple*`)` returns `True` if `str` ends with *substr-or-tuple*

```
>>> 'bang! a whimper'.endswith('a whimper')
True
```

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

https://xkcd.com/208/

Regular Expressions

In computer science, a language is a set of strings. Like any set, a language can be specified by enumeration (listing all the elements) or with a rule (or set of rules).

- A regular language is specified with a *regular expression*.
- We use a regular expression, or *pattern*, to test whether a string "matches" the specification, i.e., whether it is in the language.

Python provides regular expression matching operations in the re module.

For a gentle introduction to Python regular expressions, see Python Regular Expression How-to

Matching with `match()`

Every string is a regular expression, so let's explore the re module using simple string patterns.

`re`'s `match(`*pattern*, *string*`)` function applies a pattern to a string:

```
>>> re.match(r'foo', 'foobar')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.match(r'oo', 'foobar')
```

`match` returns a `Match` object if the string begins with the pattern, or `None` if it does not.

Notice that we use a special raw string syntax for regular expressions because normal Python strings use backslash (\) as an escape character but regexes use backslash extensively, so usgin raw strings avoids having to double-escape special regex forms that use backslash.

Finding Matches with `search()` and `findall()`

`search(`*pattern*, *string*`)` is like `match`, but it finds the first occurrence of pattern in string, wherever it occurs in the string (not just the beginning).

```
>>> re.match(r'oo', 'foobar')
>>> re.search(r'oo', 'foobar')
<_sre.SRE_Match object; span=(1, 3), match='oo'>
```

Note the `span=(1, 3)` in the returned match object. It specifies the location within the string that contained the match.

`findall` returns a list of substrings matched by the regex pattern.

```
>>> re.findall(r'na', 'nana nana nana nana Batman!')
['na', 'na', 'na', 'na', 'na', 'na', 'na', 'na']
```

The Match Object

The `match` and `search` funtions return a `Match` object. The important methods on the `Match` object are:

- `group()` returns the string matched by the regex
- `start()` returns the starting position of the match
- `end()` returns the ending position of the match
- `span()` returns a tuple containing the (start, end) positions of the match

For example:

```
>>> m.group()
'oo'
>>> m.span()
(1, 3)
>>> m.start()
1
```

Using The Match Object

Since a `match` and `search` return a `Match` object if a match is found, or `None` if no match is found, a common programming idiom is to test the `Match` object directly.

```
>>> m = re.match(r’foo’, ’foobar’)
>>> if m:
...     print(’Match found: ’ + m.group())
...
Match found: oo
```

Most of the examples in this lecture will use `findall` for simplicity and to demonstrate multiple matches in a single string.

Metacharacters

Regexes are much more powerful when you add metacharacters. We'll learn the basics of:

- `.` - Match any character
- `\` - Escape special characters
- `|` - Or operator
- `^` - Match at the beginning of a string/line
- `$` - Match at the end of a string/line
- `*` - Match 0 or more of the preceding regex
- `+` - Match 1 or more of the preceding regex
- `?` - Match 0 or 1 of the preceding regex
- `{ }` - Bounded repetition
- `[ ]` - Character class
- `( )` - Capture group within a matched substring

Patterns with Metacharacters

`.` matches any single character. This example also demonstrates that `findall` finds non-overlapping matches.

```
>>> re.findall(r'a.a', 'abracadabra')
['aca']
>>> re.findall(r'a.a', 'abra abra cadabra')
['a a', 'ada']
```

`\` escape special characters so we can match them in strings.

```
>>> re.search(r'C:\\>', '$ C:\> >>>')
<_sre.SRE_Match object; span=(2, 6), match='C:\\>'>
```

`^` and `$` match at the beginning or end of a string/line.

```
>>> re.search(r'^na', 'nana nana nana nana Batman!')
<_sre.SRE_Match object; span=(0, 2), match='na'>
>>> re.search(r'na$', 'nana nana nana nana')
<_sre.SRE_Match object; span=(17, 19), match='na'>
```

Repetition

`*` matches 0 or more of the preceding regex

```
>>> re.findall(r'a.a*', 'abra abra cadabra')
['ab', 'a a', 'a ', 'ada']
```

`+` matches 1 or more of the preceding regex

```
>>> re.findall(r'a.+a', 'abra abra cadabra')
['abra abra cadabra']
```

Notice that `.+` performed a greedy match - it matched as many characters as possible. We can make it non-greedy by adding a `?`

```
>>> re.findall(r'a.+?a', 'abra abra cadabra')
['abra', 'abra', 'ada']
```

`?` after an ordinary character matches 0 or 1 of them

```
>>> re.findall(r'ab?a', 'aba anna abba aa')
['aba', 'aa']
```

`{ }` bounds the repetition by an arbitray number

```
>>> re.findall(r'ab{2}a', 'aba anna abba abbba')
['abba']
```

Character Classes and Alternatives

`[ ]` creates an arbitrary character class

```
>>> re.findall(r’[rmpl]ain’, ’the rain in spain falls mainly in the
    plain’)
[’rain’, ’pain’, ’main’, ’lain’]
```

You can specify ranges of characters in a character class.

```
>>> re.findall(r’[0-9]+’, ’500 Tech Parkway, Atlanta, GA 30332’)
[’500’, ’30332’]
```

You can specify alternative patterns to match with `|`, which you can read as "or."

```
>>> re.findall(r’rain|plain’, ’the rain in spain falls mainly in the
    plain’)
[’rain’, ’plain’]
```

Predefined Character Classes

Character classes are useful, so several are predefined.

- `\d` Matches any decimal digit; this is equivalent to the class `[0-9]`.
- `\D` Matches any non-digit character; this is equivalent to the class `[^0-9]`.
- `\s` Matches any whitespace character; this is equivalent to the class `[ \t\n\r\f\v]`.
- `\S` Matches any non-whitespace character; this is equivalent to the class `[^ \t\n\r\f\v]`.
- `\w` Matches any alphanumeric character; this is equivalent to the class `[a-zA-Z0-9_]`.
- `\W` Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

Match Capture Groups

Capture groups allow you to match on a pattern but capture a substring of what was matched. This is particularly useful in extracting element text from XML-like documents where your pattern includes the open and close tags but you only want the text between the tags.

```
>>> activities = ’’’
... <ul>
... <li>eat</li>
... <li>sleep</li>
... <li>code</li>
... </ul>’’’
>>> re.findall(r’<li>(.+)</li>’, activities)
[’eat’, ’sleep’, ’code’]
```