Strings & Regex

- About strings
- Formatting
- HTML with Jinja2
- Slicing & Indexing
- String Built-in Methods
- Regex

About Strings

- Immutable
- Index-accessible
- "Batteries included"
- Are iterable

In [ ]:

String Formatting

We have many variable types, often we want to print their value.

In [1]:
```
print "I can print an integer %d, a string %s, or decimal %.2f" % (10, "\'string\'", 37.248)
```

```
I can print an integer 10, a string 'string', or decimal 37.25
```

Formatting Possibilities

This might seem confusing at first, but there is actually a very handy guide of these characters here (https://docs.python.org/2/library/stdtypes.html#string-formatting). Formats that you can print out natively in Python:

- Strings
- Decimals, Integer, Binary, Hex
- Boolean
- Exponential format (1.43e-7)

Formatting with Tuples

In [9]:
```
conversion = "Hexadecimal (%d): %X" % (122, 122)
print conversion
```

```
Hexadecimal (122): 7A
```

Formatting with Dictionaries

In [8]:
```
print "Item: %(name)s costs %(cost).2f. %(name)s can be sold at market"
% {
    "name" : "Rolex",
    "cost" : 4999.99,
}
```

```
Item: Rolex costs 4999.99. Rolex can be sold at market
```

New String Formatting

This is another way Python devs have come up with. It's not necessary to use it, though as you get more experienced with Python you might find instances where it's faster, syntactically speaking. Up to taste.

In [ ]:
```
fav_fruit = "apples"
fav_veggie = "broccoli"
print "I like {0} and {1}!".format(fav_fruit, fav_veggie)
```

HTML String Interpolation with Jinja2

Oftentimes in web development we'd like to output HTML in intelligent and programmatic ways. Frameworks like Django even have built-in templating systems (similar to Jinja2).

```
$ pip install jinja2
```

Making strings is as simple as:

In [14]:
```
from jinja2 import Template

template = Template("Hello {{ name }}!")
print template.render(name="Will")
```

```
Hello Will!
```

More Complex Programmatic Logic with Jinja2

In [13]:
```
from jinja2 import Template

people = [
    {'name' : 'Ray', 'number' : '324-234-3888', 'is_admin' : True},
    {'name' : 'Allie', 'number' : '415-342-3333', },
    {'name' : 'Li Wan', 'number' : '998-342-4400', },
]

template = Template("""
{% for p in people %}
    {% if not p.is_admin %}
        <p>{{ p.name }} - {{ p.number }}</p>
    {% endif %}
{% endfor %}
""")

print template.render(people=people)
```

```
        <p>Allie - 415-342-3333</p>


        <p>Li Wan - 998-342-4400</p>
```

Template Inheiritence in Jinja2

Parent template (`base.html`), basically an `abstract` class for an HTML page (to use Java parlance):

```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        © Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
</body>
</html>
```

Child template:

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
      Welcome to my awesome homepage.
    </p>
{% endblock %}
```

Slicing and Indexing

Is the same as with lists - they are both Python iterables:

```
some_string [<first=0> : <last=-1> : <stepsize=1>]
```

In [105]:
```
rhyme = "Mary had a little lamb!"

print rhyme[0]
print rhyme[-1]
print rhyme[1:18:2]
```

```
M
!
ayhdaltl
```

String Operations

In [23]:
```
sentence = "       The quick brown fox jumps over the lazy dog. \n "

print sentence.lower()
print sentence.upper()
print sentence.strip()
print sentence.split(' ')

words = sentence.strip().split()
print "Words: %s" % words
```

```
       the quick brown fox jumps over the lazy dog.

       THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.

The quick brown fox jumps over the lazy dog.
['', '', '', '', '', '', '', 'The', 'quick', 'brown', 'fox', 'jumps',
 'over', 'the', 'lazy', 'dog.', '\n', '']
       The quick brown fox jumps over the lazy pig.

Words: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
'dog.']
```

More String Operations

In [28]:
```
# join()
names = ["John", "Mary", "Roger", "Donald"]
separator = ", "
print separator.join(names)

# starts with
print names[0].startswith("Jo")
```

```
John, Mary, Roger, Donald
True
```

String `replace()`

In [30]:
```
sentence = "The quick brown fox jumps over the lazy dog."
print sentence.replace('dog', 'pig')

tougher = "Call me at 310-332-3943"

# how can we replace that phone number with another one?
```

```
The quick brown fox jumps over the lazy pig.
```

Regular Expressions in Python

Used for:

- Data cleaning
- Extracting links / phone numbers / email addresses from unstructured data
- Verifying formatting of strings
- Making programmer's lives miserable (just kidding)

`re`: Ways to Use

- `re.search()`: find a pattern in a string
- `re.match()`: does this entire string conform to this pattern?
- `re.findall()`: find ALL patterns in this string

Searching with Regex

Looks like this:

```
match = re.search(pattern, string)
```

Where match is `None` if no match, but otherwise a single Match object.

In [38]:
```
import re

sentence = "The quick brown fox jumps over the lazy fox."
pattern = r'fox'  # raw string
match = re.search(pattern, sentence)
print match

if match:
    print 'found:', match.group()
else:
    print 'did not find'
```

```
<_sre.SRE_Match object at 0x10b2691d0>
found: fox
```

Regular Expression Pattern Syntax

See here (https://docs.python.org/2/library/re.html#regular-expression-syntax), and here (https://developers.google.com/edu/python/regular-expressions?hl=en) for a little tutorial.

Basic pattern types:

- `'A'`, `'x'`, `'6'`: ordinary characters that match themselves exactly
- `'.'`: matches any character except the newline ('\n')
- `'\w'`: sequence of word-like characters [a-zA-z0-9_] that are NOT spaces
- `'\d'`: digits [0-9]
- `'\s'`: whitespace characters (space, newline, tab, etc)
- `'^'`, `'$'`: start and end of string, respectively
- `'\'`: escape char - you'll need this to match literal instances of special chars (`'.^$*+?{[]\|()'`)

Example: Joke

What do you call a pig with three eyes? Piiig!

For each below, 1) will a match be found, and 2) if so, what will match?

In [ ]:
```
match = re.search(r'iii', 'piiig')
match = re.search(r'igs', 'piiig')

## . = any char but \n
match = re.search(r'..g', 'piiig')

## \d = digit char, \w = word char
match = re.search(r'\d\d\d', 'p123g')
match = re.search(r'\w\w\w', '@@abcd!!')
```

Example: Repitition Groups

Occurences of pattern **to the left**:

- `'+'`: **1 or more**
- `'*'`: **0 or more**
- `'?'`: **0 or 1**
- `{k}`: exactly integer `k` occurences
- `{m, n}`: `m` to `n` occurences, inclusive

In [48]:
```
match = re.search(r'pi+', 'piiig')
match = re.search(r'i+', 'piigiiii')
match = re.search(r'i?', 'piigiiii')

# \s*: zero or more whitespace chars
match = re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx')
match = re.search(r'\d\s*\d\s*\d', 'xx12  3xx')
match = re.search(r'\d\s*\d\s*\d', 'xx123xx')

# ^: matches the start of string
match = re.search(r'^b\w+', 'foobar')
match = re.search(r'b\w+', 'foobar')

print match.group()
```

Excercise: Finding emails

In [58]:
```
text = "Hi Brandon,\n\nGreat meeting you the other day! Please follow up
 with gb-johnson@gmail.com for details.\n\nBest,\nAlice"
print text

# where do we start?
```

```
Hi Brandon,

Great meeting you the other day! Please follow up with gb-johnson@gmai
l.com for details.

Best,
Alice
```

Character Sets

Allows for `OR` logic in regexes.

```
[abcd]
```

matches either an `a`, `b`, `c`, or `d`.

```
[abcd]+
```

would be one or more of an `a`, `b`, `c`, or `d`.

In [66]:
```
import re

match = re.match(r'[abcd]', "a")
# match = re.match(r'[abcd]+', "abdbcbdbcbdba")

print match
print match.group()
```

```
<_sre.SRE_Match object at 0x10b29e648>
a
```

Example: Emails (pt. 2)

In [2]:
```
import re

text = "Hi Brandon,\n\nGreat meeting you the other day! Please follow up
 with gb-johnson@gmail.com for details.\n\nBest,\nAlice"
print text

pattern = "xxx"
match = re.match(pattern, text)
```

```
Hi Brandon,

Great meeting you the other day! Please follow up with gb-johnson@gmai
l.com for details.

Best,
Alice
```

Group Capture

What if our mail server only works with certain email address domains? Or we know some are spammy?

```
gb-johnson@gmail.com
```

Or domains that start in 'g'?

In [70]:
```
matching_text = "gb-johnson@gmail.com"
parts = matching_text.split("@")
name, domain = parts[0], parts[1]

print "Name: %s, Domain: %s" % (name, domain)
```

```
Name: gb-johnson, Domain: gmail.com
```

Groups in Regex

We use parenthesis to make each group:

```
( ... group1 ...) ( ... group2 ...) etc
```

and when searching:

```
match = re.search(pattern, text)
print match.group(1)
print match.group(2)
```

Example: Email (pt. 3)

In [5]:
```
text = "Hi Brandon,\n\nGreat meeting you the other day! Please follow up
 with gb-johnson@gmail.com for details.\n\nBest,\nAlice"

match = re.search(r'([\w\.-]+)@([\w\.-]+)', text)

if match:
    print match.group()   # entire string matched
    print match.group(1)  # group 1: name of email
    print match.group(2)  # group 2: domain
```

```
gb-johnson@gmail.com
gb-johnson
gmail.com
```

Multiple Matches with `findall`

In [6]:
```
multiple = "john@google.com other garbage words alice@hotmail.com someth
ing else will@yahoo.com"
pattern = r'([\w\.-]+)@([\w\.-]+)'

# findall() returns a list of tuples
matches = re.findall(pattern, multiple)

for group1, group2 in matches:
    print "Name: %s, domain: %s" % (group1, group2)
```

```
Name: john, domain: google.com
Name: alice, domain: hotmail.com
Name: will, domain: yahoo.com
```

Muliple Matches with `findall` from a File

```
with open('file.txt', 'r') as f:
    matches = re.findall(pattern, f.read())
```

Though be careful of memory concerns - this loads the entire file into memory.

Greedy vs. Non-greedy Matching with `?`

In [97]:
```
string = "#something important# and #another thing important#"
greedy_pattern = "(#.*#)"
non_greedy_pattern = "(#.*?#)"  # <-- ? is also a non-greedy modifier

# greedy
print "Greedy"
matches = re.findall(greedy_pattern, string)
for match in matches:
    print match

# not greedy
print "\nNon-greedy"
matches = re.findall(non_greedy_pattern, string)
for match in matches:
    print match
```

```
Greedy
#something important# and #another thing important#

Non-greedy
#something important#
#another thing important#
```

In [ ]:
```

```

Replacement with Regex

Allows you to replace all instances of a pattern using another pattern:

```
re.sub(pattern, replacement, original)
```

Group syntax allows you to keep intact certain parts of the original pattern that matched while changing others.

```
\<group number here>
```

In [99]:
```
multiple = "john@google.com other garbage words alice@hotmail.com someth
ing else will@yahoo.com"
email_pattern = r'([\w\.-]+)@([\w\.-]+)'
new_string = re.sub(email_pattern, r'\1@aol.com', multiple)
print new_string
```

```
john@aol.com other garbage words alice@aol.com something else will@aol.
com
```

StringIO

`StringIO` lets us hold large strings in memory, almost like an in-memory file. Why might we want this? Several reasons:

- Prevent intermediate disk access when passing file content from one function to another
- Editing / build large strings without need to write/seek disk access

So it's mostly a performance tool, and we won't dwell on it here.

In [104]:
```
import StringIO

message = "This is a cool message that could be written to a file"

file_object = StringIO.StringIO(message)
print file_object.read()
```

```
This is a cool message that could be written to a file
```

Lab: Coding Excercises

Fill in the method definitions in the file `excercises/strings.py`.

Make sure you can pass tests with:

```
$ py.test tests/test_strings.py::StringExcercises::<function_name>  # test s
ingle function
$ py.test tests/test_strings.py::StringExcercises                   # test a
ll at once
```

Wrap-Up

- Strings
  - String Operations
  - Formatting
  - HTML formatting with Jinja2
  - Slicing & Indexing
- Regex
  - Special characters
  - Repetition, character sets
  - Multiple matching
  - Replacement
  - Greedy vs. Non-greedy
- StringIO