Python Programming

Week Seven, Class One

Documenting your code

Regular Expressions

Outline

- Documenting your code
- Regular expressions
  - Character classes
  - Meta characters
  - Greedy RE's
- Object-oriented RE's in python
  - Pattern and match objects, grouping
- Testing your RE's
    - Write some strings where you WANT it to match
    - Write some strings where you DON'T want it to match

Documentation - MMI

- ***Module*** documentation
- ***Method*** documentation
- ***Instance*** variable documentation

Module documentation

- Use triple-quotes
- Add text to the top of the file
- Example:
  - `file_util.py` in `class/src`

```
"""Contains functions useful for opening files."""
```

Three part docstring for methods

- Function – what it does, its purpose
- Returns – the return type, leave blank if no return value
- Args – arguments, their type, state if they are output from another function, leave blank if there are no arguments

Example

```
def readFile(fn):
    """
    Function: Open a compressed or uncompressed file
    Returns : A file object (opened for reading)
    Args    : fn - a string, the name of a file,
                   plain text or compressed (via gzip)
    """
    import gzip
    if fn.endswith(".gz"):
        return gzip.GzipFile(fn)
    else:
        return open(fn)
```

Another (more complex) Example

- **Note**: `acc2g` is output from another function (in the same file)

```
def add_gene_id2feat(feat,acc2g):
    """
    Function: Add Entrez Gene id tags to the given feature object
    Returns :
    Args    : feat - a Mapping.Feature.KeyVal object, usually a
                     Mapping.CompoundDNASeqFeature representing an mrna
                     alignment
              acc2g - dictionary, output from get_acc2g

    Assumes that the given feature object will have an accession as the
    display_id, and that the accession would be something reported in Entrez
    Gene's gene2accession file.

    Note also that some accessions may be associated with more than one
    Entrez Gene id.
    """
```

Quiz: How can a function do useful work without return anything?

Doc strings for the entire module

- See file_util.py in class/src
  - Note the text enclosed in triple-quotes at the top of the file
  - This gets printed when you type:

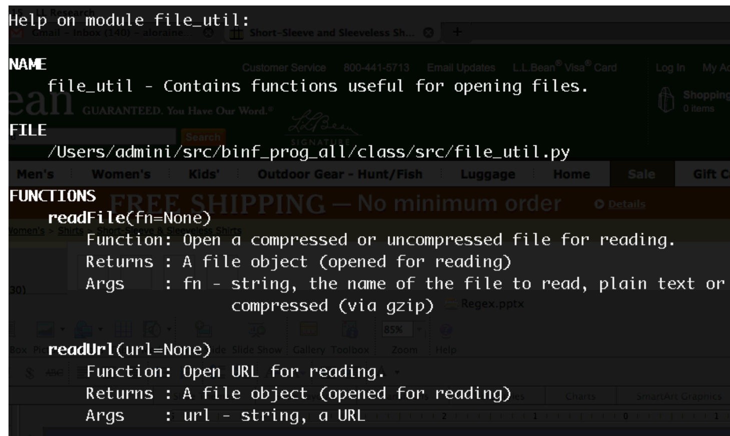
    `>>> help(modulename)` – in the interpreter

    `% pydoc modulename` – in the shell (Terminal)

Example – view doc via `help` command in the python interpreter

- Inside the interpreter:

```
>>> import file_util as f
>>> help(f)
```



Example, cont.

- In the terminal (shell), using `pydoc`

```
% pydoc file_util
```

try it

Regular Expression

- A pattern ("expression") that matches or describes a set of strings.
- Used in computing for text processing
  - Extracting instances of the pattern from larger text
- Derives from automata theory, study of abstract machines and the problems they can solve, and language theory, branches of computer science

Regular Expression (RE)

- A string that represents one or more other strings, often using meta-characters.
- In combination with the `re` module in python, lets you find instances of the string in text.
- You can use them to find indices where the matches happen, which is useful for a lot of things, such as finding restriction enzyme recognition sites in DNA.

A RE represents a ***set*** of strings

- For example:
  - AT[ATCG]AT represents
    - {AT**A**AT, AT**T**AT, AT**C**AT, AT**G**AT}
- Square brackets [] notation
  - means the RE allows any ONE of those letters to fill the middle position in the string
- RE is a **pattern** that **matches** sub-strings in text
  - We use regular expressions for **pattern-matching**.

`re` in action - python

```
>>> import re
>>> regex = re.compile(r'chr[\dCM]')
>>> s = 'chr 1 chr1 chrM chrC chr34'
>>> regex.findall(s)
['chr1', 'chrM', 'chrC', 'chr3']
```

- `[ ]` are **metacharacters**, used to define a **character class**
- `\d` is shorthand for a **character class** that matches all digits
- The rest are ordinary characters that match themselves only.

Character classes

- Defined in a regular expression using [] notation
  - `[abc]`
    - matches `a`, `b` or `c`
- Useful when matching a limited set of letters
  - `chr[1-5MC]`
    - matches chr1, chr2, chr3, chr4, chr5, chrC, chrM
  - Use hyphen to indicate a range of characters

```
>>> regex = re.compile(r'chr[1-5MC]')
>>> regex.findall('chr1 chr2 chr4 chrM chrMM')
['chr1', 'chr2', 'chr4', 'chrM', 'chrM']
```

Shorthand for character classes

- \d – any digit
- \D – any non-digit
- Guess – what do these mean?
  - \w
  - \W
  - \S
  - \s
  - \b
  - \B

Meta-characters you will use A LOT

.

Matches any non-whitespace character

^

Matches the beginning of a string

$

Matches the end of a string, before the new line, if any

RE compilation flags re.MULTILINE and re.DOTALL change these behaviors slightly. Make a guess – what do you think they do? (See section 3.5)

RE repetition meta-characters

*

Matches zero or more of the preceding character or character class

?

Matches zero or one of the preceding character or character class

+

Matches one or more of the preceding character or character class

*Sometimes also called **multipliers**, **wild cards**.*

RE greediness

- Repetitions try to match as many characters as possible
  - We say: they "consume" as much of the target string as possible before returning a match
- Example:
  - ca*t matches ct, cat, caaat, caaaaat
    - And it will!
- If later parts of the regular expression don't match, it will back up until they do.
  - Example…see next slide

Greediness in action

- Consider `a[bcd]*b` and target string `abcbd`

Step	Matched	Explanation
1	a	The a in the RE matches.
2	abcbd	The engine matches [bcd]*, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match b, but the current position is at the end of the string, so it fails.
4	abcb	Back up, so that [bcd]* matches one less character.
5	*Failure*	Try b again, but the current position is at the last character, which is a "d".
6	abc	Back up again, so that [bcd]* is only matching "bc".
6	abcb	Try b again. This time but the character at the current position is "b", so it succeeds.

From today's reading – section 2.2

Quickie Quiz

- What are these meta characters called?

```
*
?
+
```

- And what do they match?

Quick Quiz, Part II

- What is `[ ]` ?
- What do these match?

```
\d
\D
\s
\S
\b
.
```

Python RE

- In python, compile regular expressions into SRE_Pattern instances:

```
>>> import re
>>> p = re.compile(r'ab*')
>>> p
<_sre.SRE_Pattern object at 0x22e90>
```

- *Note:* "raw string" notation (`r'ab*'`)
  - This lets you use backslash characters (e.g., \d) without escaping them with more backslashes. For details, see section 3.2 of the reading "The Backslash Plague."

The compile method returns a pattern (`sre.SRE_Pattern`)

- Call methods on the pattern object to find what it matches in text

Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator.

Demo

- The `search` method returns a `Match` object, one for the first place the pattern matched.

```
>>> p = re.compile(r'\w@\S+\.edu')
>>> p.match('aloraine@uncc.edu acenglis@uncc.edu')
<_sre.SRE_Match object at 0x312f8>
```

- Note that only *one* `SRE_Match` is returned.
- Use `dir` and `help` to refresh your memory about pattern and match methods.

Demo

- To get all the Matches, use `finditer`:

```
>>> all = p.finditer('aloraine@uncc.edu
  acenglis@uncc.edu')
>>> all.next()
<_sre.SRE_Match object at 0x31090>
>>> all.next()
<_sre.SRE_Match object at 0x312f8>
>>> all.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Demo

- Use `findall` to get all the strings that matched:

```
>>> p.findall('aloraine@uncc.edu acenglis@uncc.edu')
['aloraine@uncc.edu','acenglis@uncc.edu']
```

Match object

- Has methods that let you find out exactly where the match happened:

Method/Attribute	Purpose
`group()`	Return the string matched by the RE
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple containing the (start, end) positions of the match

Also: groups() to retrieve sub-strings within a larger match

Grouping

- Sometimes you need to get part of the string that matched the RE
- Use parentheses to specific "sub-matches" and the groups command to retrieve them.
- Example:

```
>>> p = re.compile(r'(.*)\t(.*)')
>>> p.search('field0\tfield2\n').groups()[0]
'field0'
>>> p.search('field0\tfield2\n').groups()[1]
'field2'
```

Quick Quiz: What does the . (dot) character match?

Testing RE's

- For each RE you write and use:
  - Write strings you want it to match
  - Write strings you DON'T want it to match.
- And then … test them both.
- Your code will be more robust if you do.
- I have seen many bugs coming from mistaken matching because the person who wrote the RE (sometimes it was me!) wrote an RE that matched too many different strings. It's better to be conservative (and more specific) with REs, typically.

Homework

- Use it to make sure you understand the basics.
- On questions where you write regular expressions, be sure to test:
  - Strings that ***should*** match
  - Strings that ***shouldn't*** match
- If you do the above every time you use them, you'll be less likely to have problems with regular expressions.