Getting down with Markdown

- or -

Upgrading Markdown

Using Markdown

```
import markdown

…
md = markdown.Markdown(  )
md.convertFile( input=rfile, output=wfile )
```

Input is unicode.
Output is unicode.
Encoding is your problem.
Default encoding is UTF-8

You can specify the type of the generated output using the :output_format" parameter:

- **xhtml1**, **xhtml5**, **xhtml** (equavalent to **xhtml1**)
- **html4**, **html5**, **html** (equivalent to **html4**)

The defaults may change, so specify an exact version.

```
md = markdown.Markdown(  )
html = md.convert( some_string )
```

Using “codecs”

```
import codecs

rfile = codecs.open( "some_file.txt", mode="r", encoding="utf-8" )
wfile = output_file = codecs.open("some_file.html", "w",
                        encoding="utf-8",
                        errors="xmlcharrefreplace"
)
```

Files are byte stream,not text.

If using convertFile you get UTF-8 encoding 'for free'. If converting text with convert you are on your own.

The codecs module also provides EncodedFile(*handle*,*input*=, *output*=, *errors*=) for wrapping a file object.

Using Markdown Extensions

The core Markdown syntax is very basic, the awesome lives in the extensions and the extension API.

```
import markdown


…
md = markdown.Markdown(  )
md.convertFile( input=rfile, output=wfile,
                        extensions=[ 'tables', 'footnotes',] )
```

Built-in extensions are registered to Markdown by name.

http://packages.python.org/Markdown/extensions/index.html

Markdown Extensions

- **Preprocessor**
  - Process the text before Markdown-*ing*.
- **Pattern**
  - Match chunks of text using regular expressions during markdown-*ing*. (Yuck!)
- **BlockParser**
  - Matches and manipulates blocks of text during markdown-*ing*.
- **Treeprocessor**
  - Processes the serialized document (XML!) after block and pattern processing.
- **Postprocessor**
  - Processes the serialized data (string) as the last stage of markdown-*ing*.

ElementTree

Unicode!

Plugging in

```
md = markdown.Markdown(  )
```

The markdown processor instance has a collection of **ordered dictionaries** that contain the processing pipeline:

- preprocessors
- inlinePatterns
- parser.blockparsers
- treeprocessors
- postprocessors

```
ordereddict.add( key, value, '_begin' | '<..' | '..>' | '_end' )
```

http://packages.python.org/Markdown/extensions/api.html#ordereddict

Treeprocessor

```
from markdown import Extension
from markdown.treeprocessors import Treeprocessor

class OGoLinksTreeProcessor(Treeprocessor):

    def process(self, tic):
        for child in tic.getchildren():
            if child.tag == 'a':
                ...

    def run(self, doc):
        self.process( doc )
```

Wikis are all about links! So lets process all the link tags to add our own extra-special awesome.

```
class OGoLinksExtension(Extension):
    def __init__(self, configs):
        ...

    def extendMarkdown(self, md, md_globals):
        ogoext = OGoLinksTreeProcessor( md )
        ...
        md.treeprocessors.add( "ogo", ogoext, "_end" )
```

Do this as the last step of this phase

Make invalid and inaccessible links display as ~~linktext~~

Link Transform Using TreeProcessor

[This is a link](OGo#123456789)

This is a link

This is a link

8536080	Vendor: Nivel (v519)
13830101	Vendor: Homecare (vE92)
4514080	Vendor: ABI Industries (v927)
781830	Vendor: Graco (v560)
174690	Vendor: MCFA (Cat/Mitsu/NYK/Jungheinrich v348/v320/vD35)
396330	Vendor: Wise(vA05)

Using Your Custom Extension

```
import markdown
from extensions import OGoLinksExtension

...
extensions=[ 'tables', 'footnotes',
                    OGoLinksExtension( { 'context':  self.context,
                                                    ... }  ), ]
md = markdown.Markdown( extension=extensions )
```

You can register your extensions as a *string-name*, but if you only use it internally … why bother? That just obscures.

http://packages.python.org/Markdown/extensions/api.html#makeextension

Block Processor

```
from markdown.blockprocessors import BlockProcessor

class OGoQueryTableProcessor(BlockProcessor):

    def test(self, parent, block):
        rows = [ row.strip() for row in block.split( '\n' ) ]
        result =  ( len( rows ) > 2 and
                rows[ 0 ].startswith( '{OGoQueryTable{' ) and
                rows[ -1 ].endswith( '}OGoQueryTable}' ) )
        return result

    def run(self, parent, blocks):
        …
        table = etree.SubElement( parent, 'table' )
        ...
```

Blocks in markdown are primarily controlled by whitespace (indention).

Add to the parent.

Returns true or false if this block should be processed by this BlockProcessor

Again, make your extension an extension

```
class OGoQueryTableExtension(markdown.Extension):

    def __init__(self, configs):
        ...

    def extendMarkdown(self, md, md_globals):
        """ Add an instance of TableProcessor to BlockParser. """
        ext = OGoQueryTableProcessor(md.parser)
        ...
        md.parser.blockprocessors.add( 'ogoquerytable', ext, '<table ')
```

Put it in the BlockProcessor pipeline.

Where? Don't expect the documentation to help; look at examples and guess, and experiment.

print(md.parser.blockprocessors)

argparse is handy

```
{OGoQueryTable{
 entity ::kind Task
 display ::border
 column ::title "Object Id" ::alignment center ::attribute object_id ::link
 column ::title "Title" ::alignment left ::attribute name
 column ::title "Status" ::alignment center ::attribute state
 query ::key state ::expression notequals ::value 30_archived
 query ::key project_id ::expression equals ::value $__projectid__;
}OGoQueryTable}
```

```
def get_parser( )
    parser = ArgumentParser( prefix_chars=':')
    subs = parser.add_subparsers(dest="subparser")
    column_parser = subs.add_parser('column', prefix_chars=':' )
    column_parser.add_argument( '::title', action='store', type=str,  )
    ...
    return parser`
```

Using argparse on strings

```
from argparse import ArgumentParser
from shlex import split as _split
safe_split = lambda a: [ b.decode( 'utf-8' ) for b in _split( a.encode( 'utf-8' ) ) ]

def get_parser( )
  ...

class OGoQueryTableProcessor(BlockProcessor):

    def run(self, parent, blocks):
        parser = get_parser( )
        for line in blocks.pop(0).split('\n')[ 1:-1 ]:
            args = parser.parse_args( safe_split( line ) )
```

Some glue is required to use safe_split across Python's w/Unicode

Our query extension

```
{OGoQueryTable{
 entity ::kind Task
 display ::border
 column ::title "Object Id" ::alignment center ::attribute object_id ::link
 column ::title "Title" ::alignment left ::attribute name
 column ::title "Status" ::alignment center ::attribute state
 query ::key keywords ::expression ilike ::value "%ctabs-invoice%"
 query ::key project_id ::expression equals ::value $__projectid__;
}OGoQueryTable}
```

Object Id	Title	Status
81166905	Provision security context for selecting shipment data via OIE	30_archived
83820172	Add FedEx data to the shipping manifest sent to C-tabs	20_processing
79410909	Send "demo" invoices to C-Tabs	30_archived
82711432	Transform our shipment manifest into retarded 1980s vintage format	00_created
81167299	Need Worfklow Stages To Select & Transform Shipment information	30_archived

Who's your daddy?

```
class OGoQueryTableProcessor(BlockProcessor):

    def run(self, parent, blocks):
        .
```

<Element 'div' at 0x45c20f0>
- or -
<Element 'li' at 0x45c3810>
- or -
...

You can tell what you are inside of; in case you want to adapt to being nested in a table cell, part of a list, etc...

NOTE: Not all the extensions, especially table, handle nested markup well, if at all.