UF-Tools Documentation

Release 1.0

N/A

October 20, 2014

Contents

CHAPTER 1

What is Century?

Let's say you'd like to write a simple script to work with a certain page on the University of Florida website. What may seem like a simple task could very well prove quite complicated: you'll have to handle authentication, cookies, redirects, and often times broken html. When it comes down to it, UF's webpages don't play well with scripting.

Century handles automation of the UF site, and does it for you with an extensible, documented API. Now you can write that app, without worrying about where you're going to get the data from. Tasks that took a few thousand lines of code now only take a couple hundred. We're here to make what should've been an open system, open.

Please note that we are not affiliated with the University of Florida in any way. This API is unofficial, and unsupported by UF. If you manage to get in trouble while using it, it's your own fault.

CHAPTER 2

Contents

2.1 Getting Started

This guide is intended to get one started with Century. If you haven't used Century before, this is a good place to start. We'll build a couple utilities with the high-level functionality of Century, and then work our way down.

2.1.1 Contents

Installing

Dependencies

Python 3 Python 3 is an absolute no-if-ands-or-buts-about-it requirement for Century, and it is probably safe to assume you have used it before (otherwise, *why would you be reading this?*), but just to play it safe:

> **Warning:** `python3` is different than the command-line `python` included with some systems, which typically refers to Python version 2.x for compatiblity reasons.

Windows Users will need to perform a manual install, for which instructions can be found on the Python download page. Alternatively, Portable Python can be used if one does not wish to, or does not have permissions to install Python on a machine.

Mac OS X Users should upgrade to the newest version of Python 3, preferably through macports or homebrew. Instructions on installing packages through those tools can be found on their respective webpages.

Ubuntu and Debian Linux Users probably already have a version of Python 3 installed. Ubuntu includes it by default as of *11.10*. If not, this can easily be resolved by running `sudo aptitude install python3`, or, if the system does not have aptitude installed, `sudo apt-get install python3`.

Notably, (as of writing) the CISE department servers do not have Python 3 installed (as they are on the 10.04 LTS). This issue can be resolved by copying a version of StaticPython (preferably renamed to `python3`) to a `bin` folder in your home directory. (Unfortuately the later recommended LXML dependency is not so easily resolved)

Fedora and RedHat Linux Users Again, most users will probably already have Python 3 installed. If not, one can run `yum install python3` as root.

LXML The core of Century has no required dependencies besides Python 3, and so one can run Century with no extra packages. That said, almost all of the modules in `lib.tasks` need LXML in order to quickly perform their complex html processing and transformations.

Unfortunately, installing LXML is typically not as easy as installing Python 3. Some instructions for Windows and Mac users can be found on the LXML installation webpage. OS X users should additionally consult their package manager's documentation and repositories.

Ubuntu and Debian users can install LXML just like they did Python 3, by running `sudo aptitude python3-lxml`, or `sudo apt-get python3-lxml`. Fedora and RedHat users can install LXML with `sudo yum python-lxml` (which actually contains both Python 2 and 3 versions).

Sphinx Sphinx is only needed to build the documentation, and considering that you're reading it right now, you probably don't need to build it. However, if you do, it is installable on Debian and Ubuntu with `sudo aptitude install python3-sphinx` or `sudo apt-get install python-sphinx`.

Note: It is possible that the package is unavailable on your system, because (as of writing) the package is **very** new. In fact, Century 1.0's documentation had to be built using `virtualenv` and Sphinx from the pypi (`easy_install`). If this is the case, you can either try to install it through apt-pinning or by manually downloading and installing the 'deb' package from the Debian Sid package page.

Century

Now that you have the dependencies out of the way, you can get onto installing Century. Being a pure-python library, installing and using it is rather trivial. One can download the latest version of Century from the github page, or alternatively (if `git` is installed, clone the latest bleeding-edge version from github with:

```
$ git clone https://github.com/CenturyDeveloper/Century.git century
```

The `lib` folder can then be lifted directly from the repository directory, and placed into a project folder. Alternatively, instead of moving over the lib directory, the library can be placed relative to a project directory, and then `sys.path` can be manipulated to make Python search for Century in an alternative location, like so:

```
# This code should execute before Century is imported for the first time.
import sys
sys.path.append(os.path.abspath("../relative/path/to/century"))
```

Now you're ready to make your first program with Century!

Building Documentation As an aside, if you wish to build a copy of this documentation, so that you can have have a local copy of it, or so that you can be sure to have the exact version matching your installation, this can be done by `cd`'ing into the `doc` directory, and (on Linux or OS X) running:

```
$ make html
```

or on Windows:

```
$ make.bat html
```

The results will show up in `doc/_build/html/index.html`.

Alternatively, if you only want an offline copy of the documentation, it can be downloaded from the Century gh-pages branch.

Representing and Handling Course Sections

The `lib.tasks.courses` module contains various constructs for representing courses at the University of Florida.

Note: The appropriate classes from the `lib.tasks.courses` module should be imported before the code is executed.

The `CourseCode` Class

`lib.tasks.courses.CourseCode` is a subclass of `collections.UserString`, meaning it can be used like a `str` object, but also like a traditional object with various properties. When constructing a `lib.tasks.courses.CourseCode`, a code string is forced into a consistant format.

```
>>> CourseCode("mac 2311")
'MAC2311'
>>> CourseCode("phy2048 L")
'PHY2048L'
>>> CourseCode("phy 2048 l") == CourseCode("PHY2048L")
True
```

Getting the prefix of the course code is trivial, and can be done in a readable fashion:

```
>>> CourseCode("MAC2311").prefix
'MAC'
>>> CourseCode("MAC2311")[0:3] # alternatively
'MAC'
```

Various other properties and documentation can be found on the `lib.tasks.courses.CourseCode` page.

The `CourseMeeting` Class

Representing course meeting times is a step more complex from course codes. Often, one will see course meeting times represented on ISIS or the Registrar with sets of day and period identifiers, such as `MWF 2-3`. One `lib.tasks.courses.CourseMeeting` object represents one day/period pair. Each `lib.tasks.courses.Course` object can contain multiple `lib.tasks.courses.CourseMeeting` objects. Constructing one of these objects can be done a few ways:

```
>>> str(CourseMeeting("MWF", "2-3", building="MAEB", room="123C"))
'Periods 2-3 on M W F at MAEB 123C'
>>> str(CourseMeeting((Days.MONDAY, "W", "F"), (2, 3), "MAEB", "123C"))
'Periods 2-3 on M W F at MAEB 123C'
```

Once you have a `lib.tasks.courses.CourseMeeting` object, you can do some moderately interesting things with it (although it is mainly just intended as a container):

```
>>> mae_meeting = CourseMeeting("MWF", "2-3", building="MAEB", room="123C")
>>> Days.WEDNESDAY in mae_meeting.days and 3 in mae_meeting.periods
True
>>> mae_meeting.periods_str
'2-3'
```

Again, more documentation is available on the `lib.tasks.courses.CourseMeeting` page.

The `Course` Class

Now that we've covered the dependent classes, we can begin building and using `lib.tasks.courses.Course` objects.

While it might be a bit of a misnomer, a single `lib.tasks.courses.Course` object represents a single section of a class, rather than a class itself. This may seem a bit odd, but it is this way as an attempt to map things a bit closer to how ISIS and the Registrar represent their information.

The minimum requirements for a `lib.tasks.courses.Course` object are a course code, and section number (represented as a string). For convenience, a course code can be passed in as a string, and it will automatically be turned into a `lib.tasks.courses.CourseCode` object. Typically when we have a function, method, or constructor in Century that needs a course code, you can pass either a string or a `lib.tasks.courses.CourseCode` object, and conversions will happen automatically.

Let's make a simple course, for our Calc 1 (`MAC2311`) class:

```
>>> calc1 = Course("mac 2311", "145D")
>>> str(calc1)
'Course MAC2311 (Section 145D):'
```

Okay, that works, but it does feel kinda empty... From the documentation reference page for `lib.tasks.courses.Course`, we can see that the constructor will accept:

- A course title (as a `str`)
- A number of credits (as an `int`)
- A list (or any other kind of iterable) of meetings
- A one-character `str` representing the Gen-Ed credit the course gives
- A `str` specifying information about what Gordon-Rule requirements the course fulfills
- A list of instructors as strings

Note: Some processing is applied to certain attributes to fix capitalization. The processing applied to attributes is described more in depth on the class reference page.

Wow, that's a mouthful, and some of that may be a bit overkill for our needs, but fortunately we don't have to put more information in there than we want. Let's put some more information in there though:

```
>>> calc1 = Course("mac 2311", "145D", title="ANALYT GEOM & CALC 1",
...                credits=4, meetings=[mae_meeting],
...                instructors=["john doe", "ann frank", "big lequisha"])
>>> print(str(calc1))
Course MAC2311 (Section 145D):
    Title (Name): Analyt Geom & Calc 1
    Credits: 4
    Meeting: Periods 2-3 on M W F at MAEB 123C
    Taught By: John Doe
               Ann Frank
               Big Lequisha
```

Nice! We now have an object to work with, and even some "pretty" output to show for it. As is with most things in the `lib.tasks.courses` module, the `lib.tasks.courses.Course` class mainly exists as a mechanism for information storage, but there are a couple interesting things that we can do with our formed `calc1` object.

From an ISIS schedule page, one can pull up a campus map, visually showing where courses are on campus. We can generate a similar URL for one course, by fetching the `lib.tasks.courses.Course.campus_map_url` attribute:

```
>>> calc1.campus_map_url
'http://campusmap.ufl.edu/?sched=MAC2311,MWF,2-3,MAEB,123C;'
```

One can even pull up the webpage in the user's default browser using the simple `lib.tasks.courses.Course.open_campus_map()` method.

The `CourseList` Class

As a minor additional feature, you can group course objects together with `lib.tasks.courses.CourseList` objects. The class is a subclass of `collections.UserList`, so it looks, acts, behaves, and feels like a list, with a few extra features, for example, we can construct a `lib.tasks.courses.CourseList`

```
>>> calc_list = CourseList([calc1, calc1, calc1])
```

and then do:

```
>>> print(str(calc_list))
Course MAC2311 (Section 145D):
    Title (Name): Analyt Geom & Calc 1
    Credits: 4
    Meeting: Periods 2-3 on M W F at MAEB 123C
    Taught By: John Doe
               Ann Frank
               Big Lequisha
Course MAC2311 (Section 145D):
    Title (Name): Analyt Geom & Calc 1
    Credits: 4
    Meeting: Periods 2-3 on M W F at MAEB 123C
    Taught By: John Doe
               Ann Frank
               Big Lequisha
Course MAC2311 (Section 145D):
    Title (Name): Analyt Geom & Calc 1
    Credits: 4
    Meeting: Periods 2-3 on M W F at MAEB 123C
    Taught By: John Doe
               Ann Frank
               Big Lequisha
```

Notice how all the courses get pretty-printed. What if we wanted to pull up a campus map with all three of those courses? (not too interesting in our case, as all our courses are the same)

```
>>> calc_list.campus_map_url
'http://campusmap.ufl.edu/?sched=MAC2311,MWF,2-3,MAEB,123C;MAC2311,MWF,2-3,MAEB,123C;MAC2311,MWF,2-3,
```

Of course, if we can subdivide the list using slices, but what makes this slightly more interesting is that the result we get out is another `lib.tasks.courses.CourseList` object:

```
>>> len(calc_list)
3
>>> type(calc_list)
<class 'lib.tasks.courses.CourseList'>
>>> len(calc_list[0:2])
2
>>> type(calc_list[0:2])
<class 'lib.tasks.courses.CourseList'>
```

2.2 Web-Browser Services

2.2.1 `browser` – Webbrowser-like Extensable State Machine

2.2.2 Parsers and Plugins

`parsers` – Generic Webpage Parsers for the `browser` Module

Defines a set of useful parser functions, and generators for parser functions. A parser function must take three arguments:

`source` The non-decoded raw byte result from the page load. This should be the result from a call to `read()` from the urllib file hander resulting from calling urllib.

`headers` The result of a call to `info()` on the file handler.

`url` The url of the current page (after redirects).

`lib.browser.parsers.`**`beautiful_soup_html`**(*source*, *headers*, *url*)
Returns a `BeautifulSoup.BeautifulSoup` object using the BeautifulSoup library. BeautifulSoup is very error resistant, and may be useful for some rather broken html. Additionally, it's written in pure python, unlike lxml which has native dependencies. Unfortunately, it is rather slow compared to lxml, and it has poor Python 3 support at the moment.

`lib.browser.parsers.`**`beautiful_soup_xml`**(*source*, *headers*, *url*)
Returns a `BeautifulSoup.BeautifulStoneSoup` object using the BeautifulSoup library. BeautifulSoup is very error resistant, and may be useful for some rather broken xml. Additionally, it's written in pure python, unlike lxml which has native dependencies. Unfortunately, it is rather slow compared to lxml, and it has poor Python 3 support at the moment.

`lib.browser.parsers.`**`htmlparser`**(*subclass*, **args*, ***kwargs*)
Taking a subclass of `html.parser.HTMLParser` (in py3k) or `HTMLParser.HTMLParser` (in py2), or alternatively a factory returning a subclass of one of those, builds a parser. When given data, the returned parser will construct a new instance of the subclass

`lib.browser.parsers.`**`lxml_html`**(*source*, *headers*, *url*)
Returns an `lxml.etree.ElementTree()` generated with lxml's html module.

`lib.browser.parsers.`**`lxml_xml`**(*source*, *headers*, *url*)
Returns an `lxml.etree.ElementTree()` generated with lxml's etree module.

`lib.browser.parsers.`**`passthrough`**(*source*, *headers*, *url*)
Returns the byte data given my the `read()` method on the result from `urlopen`. This just returns the source argument that it's passed.

`lib.browser.parsers.`**`passthrough_args`**(*source*, *headers*, *url*)
Returns a tuple of the arguments it's given. You can then use these arguments to call multiple other parsers. Here's an example:

```
args = lib.browser.load_page(url, parser=passthrough_args)
lxml_data = lxml_html(*args)
str_data = passthrough_str(*args)
```

`lib.browser.parsers.`**`passthrough_str`**(*byte_source*, *headers*, *url*)
Like `passthrough()`, but returns a unicod `str` object rather than a sequence of bytes.

Encoding is automatically determined via http headers or (if that fails) from the html source if it has a meta http-equiv tag to handle it (assuming that tag can be decoded via a best-attempt UTF-8 decoding). If encoding cannot be determined, we just try to decode it in UTF-8, ignoring unknown characters.

Unfortunately, this function does not yet have a system like `BeautifulSoup.UnicodeDammit`, or `chardet`, which can actually build a statistical model of the page's possible encoding.

`lib.browser.parsers.`**passthrough_str_with_encoding**(*encoding*)
Creates and returns a custom version of `passthrough_str()`, utilizing a specified string encoding format, rather than attempting to automatically detect things.

`plugins` – Extending the `browser` Module

Plugins are things that add functionality to the browser. Some of them are not site-specific, and could be used in a variety of applications, such as the `lib.browser.plugins.cookies`, or the `lib.browser.plugins.keepalive` plugins. UF-specific plugins are kept within the `lib.browser.plugins.uf` package. Plugins are different from `lib.tasks`, for a few reasons:

- A plugin has direct access to more of the browser's features.
- A plugin typically does something simpler than a task.
- A plugin has to be careful about what it overrides, as it could create serious problems if poorly implemented.

If you don't know what you're doing, but you want to add more functionality to the library, I'd suggest writing a task, but for certain things, a plugin becomes a more elegant solution.

Plugins for `lib.browser.Browser` can do a variety of things, such as adding new methods to a `lib.browser.Browser` instance, adding new properties, overriding old methods or properties, adding handlers, and adding headers to future HTTP requests. The `lib.browser.plugins.decorators` module provides some useful decorators for making this happen.

The Plugin Architechure

Knowing the ins-and-outs of this module is probably not very useful, but it is probably good to have a basic understanding of how the `lib.browser.Browser`'s plugin system works.

class `lib.browser.plugins.`**BaseBrowserPlugin**
A `lib.browser.Browser` plugin is defined as a set of patches to be applied to a Browser object. This class defines a base for browser plugins.

__init__()
Sets up everything the plugin needs. Subclasses should be sure to call this.

overrides
A list of methods overriden by this plugin instance.

extensions
A list of methods adding to the browser's internal list of methods, but not overriding pre-existing methods. Note that if two functions attempt to extend with methods of the same name, `Pluggable` will throw an exception.

property_extensions
Like `extensions`, but refering to properties instead.

handlers
A list of objects extended from `urllib.request.BaseHandler` to add to the browser's interal `urllib.request.OpenerDirector` instance automatically.

addheaders
A list of headers to add to the browser's internal `urllib.request.OpenerDirector` instance automatically, using `urllib.request.OpenerDirector.addheaders()`

_get_instance_functions()
Returns the object's list of methods, useful for the `_load_list()` method.

_load_list(*marker*)
Finds all methods with a specific marker and returns them

Keyword arguments:

marker A string representing the marker to look for on each function. If the value of this marker is *True*, the value is returned.

class lib.browser.plugins.**Pluggable**(**plugins*)
Takes in plugins, allowing one to configure an object on the fly, with a strutured monkey-patching like system

__init__(**plugins*)
Sets up everything the object needs, and adds plugins specified by the positional arguments (they can be added later too). Subclasses should be sure to call this.

plugins
The list of plugins already loaded into the `Pluggable`.

_register_plugin_attribute(*name*, *handler*)

__getattr__(*name*)
Used to intercept getting attributes with the `Pluggable` instance, translating them to alternate "extended" attributes.

__setattr__(*name*, *value*)
Used to intercept setting attributes with the `Pluggable` instance, translating them to alternate "extended" attributes.

load_plugins(**plugins*)
Loads in one or a group of plugins. Rather than overriding this, subclasses should use the `lib.browser.plugins.decorators.plugin_attribute()` and `lib.browser.plugins.decorators.plugin_attribute_handler()` decorators. Attributes are loaded from plugins, and then parsed to be used in extending the `Pluggable` subclass' object.

overrides(*name*, *overriding_function*)
The handler for the `lib.browser.plugins.decorators.override()` decorator. `name` is the name of the property it is set to override, `overriding_function` is the function that we should set to that name. It should take at least 2 arguments (in addition to `self`, which would refer to the plugin instance), the browser instance and the function object that is being overridden.

extensions(*name*, *extending_function*)
The handler for the `lib.browser.plugins.decorators.extension()` decorator. `name` is the name of the property it is set to extension, `extending_function` is the function that we should set to that name. It should take the browser instance as an argument in addition to `self`, which would refer to the plugin instance. Any additional arguments are passed through.

property_extensions(*name*, *value*)
The handler for the `lib.browser.plugins.decorators.property_extension()` decorator. Adds a virtual property with the given `name`, by calling `value` with the browser instance and recieving a property object.

extensions(*name*, *extending_function*)
The handler for the `lib.browser.plugins.decorators.extension()` decorator. `name` is the name of the property it is set to extension, `extending_function` is the function that we should set to that name. It should take the browser instance as an argument in addition to `self`, which would refer to the plugin instance. Any additional arguments are passed through.

load_plugins(**plugins*)
Loads in one or a group of plugins. Rather than overriding this, subclasses should use the `lib.browser.plugins.decorators.plugin_attribute()` and `lib.browser.plugins.decorators.plugin_attribute_handler()` decorators. Attributes are loaded from plugins, and then parsed to be used in extending the `Pluggable` subclass' object.

overrides(*name*, *overriding_function*)
The handler for the `lib.browser.plugins.decorators.override()` decorator. `name` is the name of the property it is set to override, `overriding_function` is the function that we should set to that name. It should take at least 2 arguments (in addition to `self`, which would refer to the plugin instance), the browser instance and the function object that is being overridden.

property_extensions(*name*, *value*)
The handler for the `lib.browser.plugins.decorators.property_extension()` decorator. Adds a virtual property with the given `name`, by calling `value` with the browser instance and recieving a property object.

`decorators` – Decorators for Writing Plugins and Extending `Pluggable`

A set of decorator functions designed to make building a plugin or extending `lib.browser.plugins.Pluggable` a snap. If you're writing either, this is the API you'll probably work with the most.

Plugin Related Decorators

`lib.browser.plugins.decorators.`**override**(*f*)
When given a function, adds an attribute to the function, `is_override`. An overriding function should take 3 arguments, `self` (or `plugin`), `browser`, and an argument representing the function they are overriding.

```
@override
def load_page(plugin, browser, old_function, *args, **kwargs):
    # do something here
    return old_function(*args, **kwargs)
```

`lib.browser.plugins.decorators.`**extension**(*f*)
When given a function, adds an attribute to the function, `is_extension`. An extending function should take 2 arguments, `self` (or `plugin`) and `browser`.

```
@extension
def new_functionality(plugin, browser, first_argument, second_argument):
    # do something here
    pass
```

`lib.browser.plugins.decorators.`**property_extension**(*f*)
When given a function, adds an attribute to the function, `is_property_extension`. The function should take 2 arguments, `self` (or `plugin`) and `browser`. It should return an instance of `property`.

```
@property_extension
def tribbles_count(plugin, browser):

    number = 9001

    def getter():
        return number

    def setter(value):
```

```
            assert value > 9000
            number = value

        return property(getter, setter)
```

`Pluggable` Related Decorators

`lib.browser.plugins.decorators.`**`plugin_attribute`**(*f*)
A decorator which makes the passed function a handler for an attribute type with it's name. A handler should take both name and value arguments, or just a value argument if the plugin's attribute is just a list.

`lib.browser.plugins.decorators.`**`plugin_attribute_handler`**(*name*)
A decorator factory, allowing you to set multiple handler functions for the same attribute type. A handler should take both name and value arguments, or just a value argument if the plugin's attribute is just a list.

`redirect` – Base Plugin Class for Handling Redirections

`cookies` – Automatic Handling of Browser Cookies

`useragent` – HTTP `user-agent` Spoofing

class `lib.browser.plugins.useragent.`**`UserAgentSpoofer`**(*agent_string*)
Allows one to easily change the browser's HTTP `user-agent` header. This can be handy if you want to ensure that we are treated like a real-world browser.

`__init__`(*agent_string*)
Creates a new instance of the plugin using the specified user-agent string. The string can be custom, or can be pulled from one of the class' example strings.

`lib.browser.plugins.useragent.`**`firefox`**
A dictionary containing user-agent strings for `iceweasel-linux-5.0`, `firefox-macintosh-5.0`, and `firefox-windows-4.0`.

`keepalive` – Handler for HTTP `Keep-Alive` Headers

`keepalive`

`keepalive.handler`

`plugins.uf` – UF-Specific Plugins

While it is a great idea to solve problems in the most general-case form possible, that approach isn't always best. This package is about taking a pragmatic view on things, filling in the gaps where generic `browser` plugins just won't cut it, and probably never will.

`login` – Automatic and Semi-Automatic Gatorlink Login Handling

isis – Tools for Working With ISIS (Integrated Student Information System)
class lib.browser.plugins.uf.isis.**IsisBrowserTools**
Contains various simple ISIS utilities.

__init__()

load_isis_page(*plugin*, *browser*, *page_code*, **args*, ***kwargs*)
Given a page code (given by the links on the isis sidebar), loads a page. Examples of this are TRQ-SPEND, RSI-GRADES or RSI-RGHOLD. Codes appear to be in all caps, composed of 3 letters, followed by a hyphen, and then another small group of letters, however this could change. Page codes can be submitted as either HTTP GET or POST requests, however this method simply uses GET requests, making requests apparent when looking at the loaded URL.

2.3 Task-Oriented Services

While the browser-oriented features of Century in the browser module and its sub-modules provides generic services for dealing with UF's site (and potentially even other websites), the tasks module serves to accomplish specific goals, such as pulling a student's home address from the UF phonebook (after proper authentication, of course), or looking up a student's schedule (via ISIS, with that student's GatorLink).

The point here is to build up a pool of ready-to-go components that can be used in larger applications without much fuss. Furthermore, tasks are not restricted to the browser.Browser namespace, and so rules governing their development are not as strict. For example, there are no limitations on dependencies here (within reason), while everything in browser is intended to be usable with nothing but the standard Python library. The point is that you can selectively choose which parts of Century you want to use, and pay the cost of extra dependencies *as you go*.

2.3.1 Pre-Built Tasks

courses – Object Representations of UF Course Information

courses.fuzzy_match – Fuzzy Matching with List and String-like Objects

Algorithms All algorithms share one property; that they must be callable in the form:

```
algorithm(s1, s2)
```

Where s1 and s2 are the two objects to compare.

Meta-Algorithms Meta-Algorithms are functions that use other algorithms in various ways.

phonebook – Handling of the UF Student and Faculty Directory

Supporting Modules

person – Object Representations of a Person's Directory Listing

fields – Constructs to Aid Handling of Phonebook Field Types
info_dict
Contains the global list of defined fields. Given a name, one should be able to pull instrospection information from here.

http – A Phonebook Web-Interface Backend

ldap.utils – Processing for LDAP-Like Person Listings A set of shared utilities useful for dealing with LDAP or LDAP-like data sources. Contains a list of LDAP-entry defined fields and a partial processor for LDAP data.

lib.tasks.phonebook.ldap.utils.**supported_fields**
A set of fields supported by the `process_data()` function.

address (__builtin__.str) The person's home address (or if that's not available, their first available physical address).

affiliation (__builtin__.str) Described here. As of writing, possible values include "faculty", "staff", "student", and "member".

birth_date (datetime.date) An instance of `datetime.date` representing the person's date of birth.

department_number (__builtin__.str) A code representing what department this person belongs to.

email (__builtin__.str) The person's email address.

employee_number (__builtin__.str) If this person is employed by the University of Florida, this is their employee number. This is the same as one's UFID Number.

gatorlink (__builtin__.str) The person's gatorlink username.

gatorlink_email (__builtin__.str) The person's gatorlink email address (if not explicitly provided, this is guessed to be the person's gatorlink, with "@ufl.edu" appended onto the end).

language (__builtin__.str) The person's prefered language.

name (__builtin__.str) The person's name, typically: "Lastname, Firstname"

office_address (__builtin__.str) The person's office address (if available).

phone (__builtin__.str) The preferred phone number (or at least the first one that shows up

preferred_phone (__builtin__.str) An alias to `phone`.

raw_ldap (__builtin__.dict) The raw fields pulled from the person's LDAP entry.

title (__builtin__.str) Usually something like "student", or "Resident, DN-ORAL SURGERY RESIDENT"

lib.tasks.phonebook.ldap.utils.**process_data**(*data*)
Takes a dict or list of tuples with "raw" LDAP data in string format, and processes it into more pythonic objects with keys matching those listed in `supported_fields`. The processed dict is returned.

tasks.isis – ISIS-Related Tasks and Utilities

Submodules

schedule_reader – Tools for Reading a Schedule from ISIS

tasks.registrar – Registar-Related Tasks and Utilities

Submodules

course_listings – Pulling Course Data from the Registrar

2.4 Indices and tables

- *genindex*
- *modindex*
- *search*

l

Symbols

A

B

E

F

H

I

L

O

P

S

U