Sistemas Web Desconectados

Release 1

van Haaster, Diego Marcos; Defossé, Nahuel

August 11, 2009

Índice general

Índice:

CAPÍTULO 1

Tecnologías del servidor

1.1 CGI

CGI es bla

1.2 WSGI

WSGI es CGI para Python.

1.3 Lenguajes interpreteados

Python es un lengauje interpretado.

1.4 Frameworks web

Un framework web es una cosa loca.

1.5 Django

Django es un framework web escrito en Python el cual sigue vagamente el concepto de Modelo Vista Controlador. Ideado inicialmente como un adminsitrador de contenido para varios sitios de noticias, los desarrolladores encontraron que su CMS era lo sufcientemente genérico como para curbir un ámbito más aplio de aplicaciones.

En honor al músico Django Reinhart, fue liberado el código base bajo la licencia *BSD* en Julio del 2005 como Django Web Framework. El slogan del framework fue "Django, Él framework para perfeccionistas con fechas límites" [1].

En junio del 2008 fue anuncidada la cereación de la Django Software Fundation, la cual se hace cargo hasta la fecha del desarrollo y mantenimiento.

[1] Del ingles "The Web framework for perfectionists with deadlines"

Los orígenes de Django en la administración de páginas de noticias son evidentes en su diseño, ya que proporciona una serie de características que facilitan el desarrollo rápido de páginas orientadas a contenidos. Por ejemplo, en lugar de requerir que los desarrolladores escriban controladores y vistas para las áreas de administración de la página, Django proporciona una aplicación incorporada para administrar los contenidos que puede incluirse como parte de cualquier proyecto; la aplicación administrativa permite la creación, actualización y eliminación de objetos de contenido, llevando un registro de todas las acciones realizadas sobre cada uno (sistema de logging o bitácora), y proporciona una interfaz para administrar los usuarios y los grupos de usuarios (incluyendo una asignación detallada de permisos).

Con Django también se distribuyen aplicaciones que proporcionan un sistema de comentarios, herramientas para sindicar contenido via RSS y/o Atom, "páginas planas" que permiten gestionar páginas de contenido sin necesidad de escribir controladores o vistas para esas páginas, y un sistema de redirección de URLs.

Django como framework de desarrollo consiste en un conjunto de utilidades de consola que permiten crear y manipular proyectos y aplicaciones.

1.5.1 Estructuración de un proyecto en Django

El framework parte de la base que un proyecto está compuesto por un conjunto de aplicaciones.

Un proyecto es un paquete que contiene 3 módulos:

- **manage.py** Interfase de consola para la ejecución de comandos
- **urls.py** Mapeo de URLs en vistas (funciones)
- **settings.py** Configuración de la base de datos, directorios de plantillas, etc.

Durante la instalación del framework en la computradora del desarrollador, se añade al path un comando con el nombre django-admin.py. Mediante este comando se crean proyectos basados en Django.

```
$ django-admin.py startproject mi_proyecto # Crea el proyecto mi_proyecto
```

El proyecto funciona como un contenedor de aplicaciones que ser rigen bajo la misma base de datos, los mismos templates, las mismas clases de middleware entre otros parámetros.

Una aplicación es un paquete que contiene al menos los módulos:

- **models.py** Definición de entidades para el mapaeador objeto-relacional
- **views.py** Definición de las funciones que generan contenido al cliente

Y si bien, el comadno **startapp** no lo genera, se eventualmente se termina agregando un módulo:

- **urls.py** Mapeo de URLs a funciones (del módulo views de la aplicación)

El comando django-admin permite crear proyectos, su sintaxis es la siguiente:

```
$ django-admin.py startproject mi_proj # Crear un proyecto
```

La estructura de archivos creada es la siguiente:

```
mi_proyecto
|-- __init__.py
|-- manage.py
|-- settings.py
`-- urls.py
```

A continuación veremos que utilidad tiene cada uno de los módulos en el proyecto.

Módulo settings

Este módulo define la configuración del proyecto, siendo sus atributos principales la configuración de la base de datos a utilizar, la ruta en la cual se encuentran los médios estáticos, cuál es el nombre del archivo raíz de urls (generalmente urls.py). Otros atributos son las clases middleware, las rutas de los templates, el idioma para las aplicaciones que soportan *i18n*, etc.

Al ser un módulo del lengauje python, la confguración se puede editar muy facilmente a diferencia de configuraciones realizadas en XML, además de contar con la ventaja de poder configurar en caliente algunos parametros que así lo requieran.

Un parametro fundamental es la lista denominada INSTALLED_APPS que contiene los nombres de las aplicaciones instaladas en le proyecto.

Módulo manage

Esta es la interfase con el framework. Éste módulo es un script ejecutable, que recibe como primer argumento un nombre de comando de django.

Los comandos de django pemiten, permiten entre otras cosas:

- **startapp <nombre de aplicación>** Crear una aplicación
- **runserver** Correr el proyecto en un servidor de desarrollo.
- **syncdb** Generar las tablas en la base de datos de las aplicaciones instaladas

El resultado de el comando **startapp** en el ejemplo anterior genera el siguiente resultado:

```
mi_proyecto
|-- mi_aplicacion
|   |-- __init__.py
|   |-- models.py
|   |-- tests.py
|   `-- views.py
|-- __init__.py
|-- manage.py
|-- settings.py
`-- urls.py
```

Módulo urls

Este nombre de módulo aparece a nivel proyecto, pero también puede aparecer a nivel aplicación. Su misión es definir las asociaciones entre URLs y vistas, de manera de que el framework sepa que vista utilizar en función de la URL que está requiriendo el clinete. Las URLs se ecriben mediante expresiones regulares. Se suele aprovechar la posibilidad del modulo de expresiones regulares del lenguaje python, que permite recuperar gurpos nombrados (en contraposición al enfoque ordinal tradicional).

La asociación url-vistas se define en el módulo bajo el nombre *urlpatterns*. También es posible derivar el tratado de una parte de la expresión reuglar a otro módulo de urls. Generalmente esto ocurre cuando se desea delegar el tratado de las urls a una aplicación particular.

Ej: Derivar el tratado de todo lo que comience con la cadena personas a al módulo de urls de la aplicación personas.

```
(r'^personas', include('mi_proyecto.personas.urls'))
```

1.5.2 Estructura de una aplicación Django

Una aplicación es un paquete python que consta de un módulo models y un módulo views.

Módulo models

Cada vez que se crea una aplicación, se genera un módulo models.py, en el cual se le permite al programador definir modelos de objetos, que luego son transformados en tablas relacionales [2].

Módulo views

Cada aplicacion posee un módulo views, donde se definen las funcinoes que atienden al cliente y son activadas gracias a el mapeo definido en el módulo urls del proyecto o de la aplicación.

Las funciones que trabajan como vistas deben recibir como primer parámetro el request y opcionalmente parámetros que pueden ser recuperados del mapeo de urls.

Dentro del módulo de urls

```
# Tras un mapeo como el sigunete
(r'^persona/(?P<id_persona>\d)/$', mi_vista)
# la vista se define como
def mi_vista(request, id_persona):
        persona = Personas.objects.get(id = id_persona)
        datos = {'persona':persona, }
        return render_to_response('plantilla.html', datos)
```

El cliclo de una petición

Cada vez que un browser realiza una petición a un proyecto desarrollado en django, la petición HTTP pasa por varias capas.

Inicialmente atraviesa los Middlewares, en la cual, el middleware de Request, empaqueta las variables del request en una instancia de la calse Request.

Luego de atravesar los middlewares de request, mediante las definciones de URLs, se selecciona la vista a ser ejecutada.

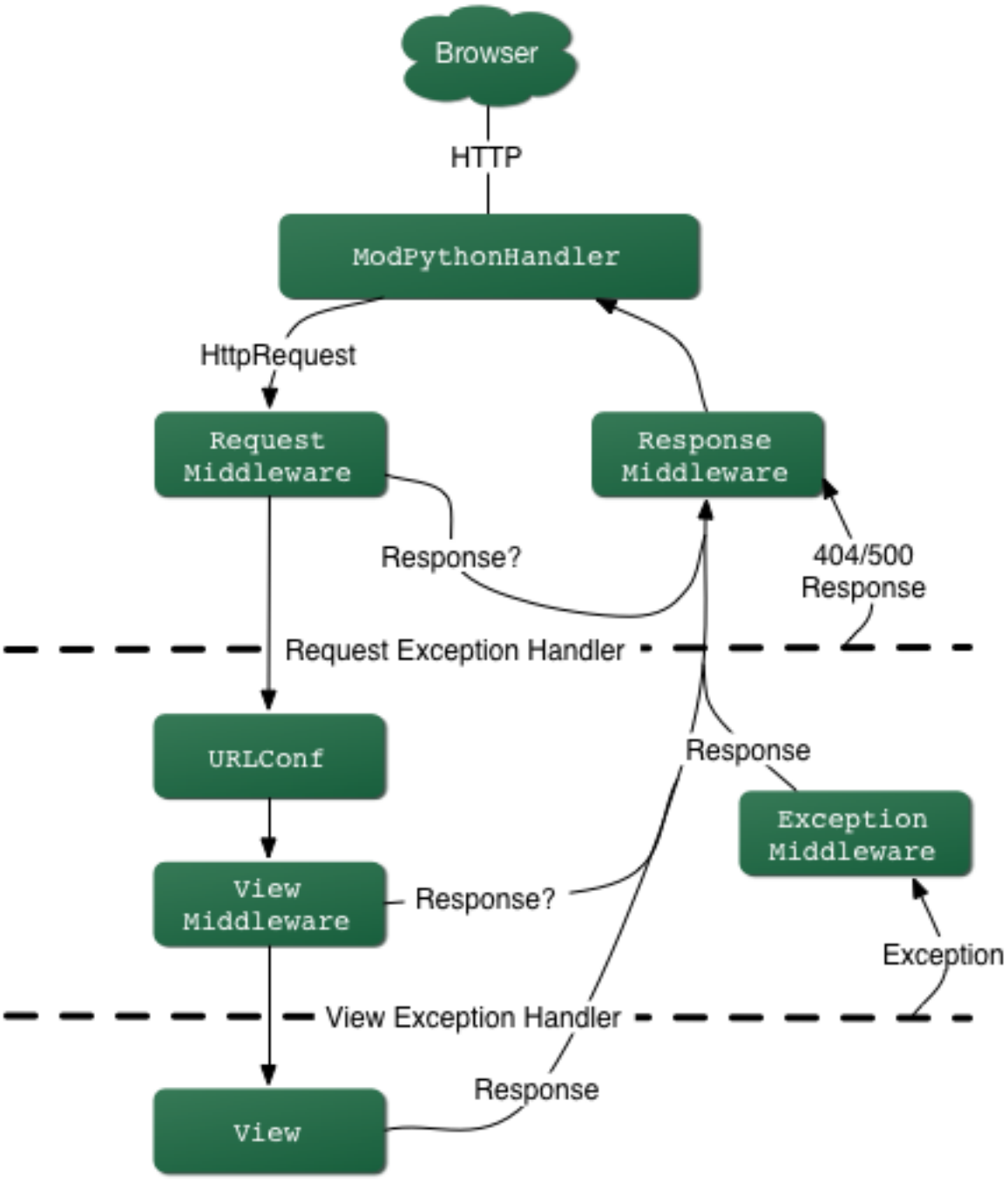
Una vista es una función que recibe como primer argumento el request y opcionalmente una serie de parámetros que puede recuperar de la propia URL.

Dentro de la vista se suelen hacer llamadas al ORM, para realizar consultas sobre la base de datos. Una vez que la vista a completado la lógica, genera un mapeo que es transferido a la capa de templates.

El template rellena sus comodines en función de los valores del mapeo que le entrega la vista. Un template puede poseer lógica muy básica (bifurcaciones, bucles de repetición, formateo de datos, etc).

El template se entrega como un HttpResponse. La responsabilidad de la vista es entregar una instancia de esta clase.

[2] Mediante el comando syncdb del módulo manage del proyecto



El Mapeador Objeto-Relacional de Django

1.5.3 Modelos

Los modelos son la fuente de información sobre los datos de la aplicacion. Esencialmente estan compuestos de campos y comportamiento propio de los datos almacenados. Generalmente, un modelo se corresponde con una tabla en la base de datos.

Dentro de un proyecto los modelos se definen por aplicacion en el modulo models.py.

Un modelo es una clase Python que hereda de `django.db.models.Model` y cada atributo representa un campo requerido por el modelo de datos de la aplicación. Con esta informacion Django genera automaticamente una *API* de acceso a los datos en la base.

Este modelo de ejemplo define una `Persona` que encapsula los datos correspondientes al `nombre` y el `apellido`.

```
from django.db import models

class Persona(models.Model):
    nombre = models.CharField(max_length = 30)
    apellido = models.CharField(max_length = 30)
```

`nombre` y `apellido` son atributos de clase

```
CREATE TABLE miapp_persona (
    "id" serial NOT NULL PRIMARY KEY,
    "nombre" varchar(30) NOT NULL,
    "apellido" varchar(30) NOT NULL
);
```

1.5.4 Consultas

bala

1.5.5 Administradores de consultas

Estos objetos representan la interfase de comunicacion con la base de datos. Cada modelo tiene por lo menos un administrador para acceder a los datos almacenados.

Glosario

API Application-Programming-Interface; conjunto de funciones y procedimientos (o métodos, si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

DOM Document-Object-Model; interfaz de programación de aplicaciones que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos.

JSON JavaScript-Object-Notation; formato ligero para el intercambio de datos.

RPC Remote-Procedure-Call; es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos.

field An attribute on a *model*; a given field usually maps directly to a single database column.

generic view A higher-order *view* function that abstracts common idioms and patterns found in view development and abstracts them.

model Models store your application's data.

MTV hola

MVC Model-view-controller; a software pattern.

project A Python package – i.e. a directory of code – that contains all the settings for an instance of Django. This would include database configuration, Django-specific options and application-specific settings.

property Also known as "managed attributes", and a feature of Python since version 2.2. From the property documentation:

> Properties are a neat way to implement attributes whose usage resembles attribute access, but whose implementation uses method calls. [...] You could only do this by overriding `__getattr__` and `__setattr__`; but overriding `__setattr__` slows down all attribute assignments considerably, and overriding `__getattr__` is always a bit tricky to get right. Properties let you do this painlessly, without having to override `__getattr__` or `__setattr__`.

queryset An object representing some set of rows to be fetched from the database.

slug A short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs. For example, in a typical blog entry URL:

```
http://www.djangoproject.com/weblog/2008/apr/12/spring/
```

the last bit (`spring`) is the slug.

template A chunk of text that separates the presentation of a document from its data.

view A function responsible for rending a page.

BSD ve ese de

i18n La internacionalización es el proceso de diseñar software de manera tal que pueda adaptarse a diferentes idiomas y regiones sin la necesidad de realizar cambios de ingeniería ni en el código. La localización es el proceso de adaptar el software para una región específica mediante la adición de componentes específicos de un locale y la traducción de los textos, por lo que también se le puede denominar regionalización. No obstante la traducción literal del inglés es la más extendida.

CAPÍTULO 3

Indices, glosario y tablas

- *Index*
- *Module Index*
- *Glosario*

chapter{Protopy} label{ch:apendiceProtopy} Protopy es una librería en JavaScript que simplifica el desarrollo de aplicaciones web dinámicas. Agregando un enfoque modular para la inclusión de código, orientación a objetos, soporte para AJAX, manipulación del DOM y eventos.

section{Módulos} Uno de los principales inconvenientes a los que Protopy da solución es a la inclusión dinámica de funcionalidad bajo demanda, esto es logrado mediante los módulos. Básicamente un modulo en un archivo con código javascript que recide en el servidor y es obtenido y ejecutado en el cliente.

begin{lstlisting}[style=javascript,label=estructura-modulo,caption=Estructura de un modulo] //Archivo: tests/module.js require('event');

var h1 = $('titulo');

function set_texto(txt) { h1.update(txt);

}

function get_texto() { return h1.innerHTML;

}

event.connect($('titulo'), 'click', function(event) { alert('El texto es: ' + event.target.innerHTML);

});

publish({ set_texto: set_texto, get_texto: get_texto

}); end{lstlisting}

begin{lstlisting}[style=consola] >>> require('tests.module') >>> module.get_texto() "Test de modulo" >>> module.set_texto('Un titulo') >>> require('tests.module', 'get_texto') >>> get_texto() "Un titulo" >>> require('tests.module', '*') >>> set_texto('Hola luuu!!!') >>> get_texto() "Hola luuu!!!" end{lstlisting}

section{Módulos incluidos} Estos módulos están incluidos en el núcleo de Protopy, es decir que están disponibles con la sola inclusión de la librería en el documento. Los módulos que a continuación se detallan engloban las herramientas básicas requeridas para el desarrollo del lado del cliente. subsection{builtin} Este modulo contiene las funciones principales de Protopy, en el se encuentran las herramientas básicas para realizar la mayoría de las tareas. No es necesario

requerir este modulo en el espacio de nombres principal (‘‘window’‘), ya que su funcionalidad esta disponible desde la carga de Protopy en el mismo.

paragraph{Funciones} subsubsection*{publish} begin{verbatim}

publish(simbols: Object)

end{verbatim} Publica la funcionalidad de un modulo. Para interactuar con el código definido en un modulo es necesario exponer una interfase de acceso al mismo, de esto se encarga la función publish.

subsubsection*{require} begin{verbatim}

require(module_name: String[, simbol: String...]) -> module: Object | simbol

end{verbatim} Importa un modulo en el espacio de nombres. Al invocar a esta función un modulo es cargado desde el servidor y ejecutado en el cliente, la forma en que el modulo se presenta en el espacio de nombres depende de la invocación. begin{itemize}

item {var cntx = require(‘doff.template.context’)} Importa el modulo ‘doff.template.context’ y lo retorna en cntx, dejando también una referencia en el espacio de nombres llamado ‘context’, esta dualidad en la asociación del modulo permite importar módulos sin asociarlos a una variable, simplemente alcanza con asumir que la parte final del nombre es la referencia a usar.

item{var cur = require(‘gears.database’, ‘cursor’)} Importa el modulo ‘gears.databse’ y retorna en cur el objeto publicado bajo el nombre de cursor, similar al caso anterior una referencia se define en el espacio de nombres para cursor. item{require(‘doff.db.models’, ‘model’, ‘query’)}

Importa el modulo ‘doff.db.models’ y define en el espacio de nombres las

referencias a model y query usando los mismos nombres. item{require(‘doff.core.urlpattern’, ‘*’)} Importa del modulo ‘doff.core.urlpattern’ todos los objetos publicados y los publica en el espacio de nombres.

end{itemize}

subsubsection*{type} begin{verbatim}

type(name: String, [bases: Array] [, class: Object], instance: Object) ->

Type end{verbatim} Función encargada de la construcción de nuevos tipos de objeto o simplemente clases. Una vez definido un nuevo tipo este puede ser utilizado para la construcción de instancias mediante el operador ‘‘new’‘. Los argumentos para la funcion ‘‘type” son, el nombre para el nuevo tipo de objeto, los tipos base de los cuales se hereda funcionalidad, opcionalmente los atributos y/o metodos de clase y los atributos y/o metodos para la instancia. La función que inicializa los objetos tiene por nombre verb|__init__| y es llamada en el momento de la construcción; en conjunto con otros metodos que se mencionaran a lo largo del texto estas funciones resultan de especial interés para interactuar con nuestros objetos y existen operadores en Protopy para manejarlas; esto es, no debieran ser invocadas o llamadas directamente. begin{lstlisting}[style=javascript,label=definicion-de-tipos,caption=Definicion de tipos] var Animal = type(‘Animal’, object, {

contador: 0,

}, { __init__: function(especie) { this.especie = especie; this.orden = Animal.contador++;

}

});

var Terrestre = type(‘Terrestre’, Animal, { caminar: function() { console.log(this.especie + ‘ caminando’);

}

});

var Acuatico = type(‘Acuatico’, Animal, { nadar: function() { console.log(this.especie + ‘ nadando’);

}

});

var Anfibio = type('Anfibio', [Terrestre, Acuatico]);

var Piton = type('Piton', Terrestre, { __init__: function(nombre) { super(Terrestre, this).__init__(this.__name__); this.nombre = nombre;

}, caminar: function() {

throw new Exception(this.especie + ' no camina');

}, reptar: function() {

console.log(this.nombre + ' la ' + this.especie.toLowerCase() + ' esta

reptando'); }

});

var doris = new Piton('Doris'); var ballena = new Acuatico('Ballena'); var rana = new Anfibio('Rana'); end{lstlisting} begin{lstlisting}[style=consola] >>> doris window.Piton especie=Piton orden=0 nombre=Doris __name__=Piton >>> rana window.Anfibio especie=Rana orden=2 __name__=Anfibio >>> isinstance(rana, Terrestre) true >>> isinstance(doris, Animal) true >>> issubclass(Anfibio, Acuatico) true >>> issubclass(Piton, Animal) true >>> doris.caminar() Exception: Piton no camina args=[1] message=Piton no camina >>> doris.reptar() Doris la piton esta reptando end{lstlisting}

subsubsection*{$} begin{verbatim}

$(id: String) -> HTMLElement $(id: String[, id...]) -> [HTMLElement...]

end{verbatim} Esta función recibe una cadena de texto y retorna el elemento del documento cuyo ''id" se corresponda con la cadena. En conjunto con la función $$ constituyen dos herramientas muy útiles para recuperar elementos e interactuar con el árbol DOM. Si mas de un argumento es pasado, la forma de retorno es mediante un arreglo, permitiendo así la iteración sobre los mismos. begin{lstlisting}[style=consola] >>> $('content') >>> $('content body') >>> $('content', 'body') [div#content, div#body] >>> $('content', 'body', 'head') [div#content, div#body, undefined] end{lstlisting}

subsubsection*{$$} begin{verbatim}

$$(cssRule: String) -> [HTMLElement...]

end{verbatim} Recupera elementos del documento, basando las reglas de seleccion en las reglas de css o hoja de estilos. begin{lstlisting}[style=consola] >>> $$('div') [div#wrap, div#top, div#content, div.header, div.breadcrumbs, div.middle, div, div.right, div#clear, div#footer, div#toolbar] >>> $$('div#toolbar') [div#toolbar] >>> $$('div#toolbar li') [li, li.panel, li.panel, li, li] >>> $$('div#toolbar li.panel') [li.panel, li.panel] >>> $$('a:not([href~=google])') [a add_post, a add_tag, a removedb, a syncdb] >>> $$('a:not([href=google])') [a add_post, a add_tag, a#google www.google.com, a removedb, a syncdb] >>> $$('div:empty') [div#logger.panel, div#dbquery.panel, div#clear, div#top] end{lstlisting}

%extend subsubsection*{extend} begin{verbatim}

extend(destiny: Object, source: Object) -> alteredDestiny: Object

end{verbatim} Extiende sobre un objeto destino todos los objetos pasados como argumentos a continuación, copiando cada uno de los atributos correspondientes, el objeto destino es retornado modificado. begin{lstlisting}[style=consola] >>> a = {perro: 4} >>> b = {gato: 4} >>> c = extend(a, b) >>> c Object perro=4 gato=4 >>> a Object perro=4 gato=4 >>> b Object gato=4 end{lstlisting}

%super subsubsection*{super} begin{verbatim}

super(type: Type, instance: Object) -> boundedObject: Object

end{verbatim} Enlaza un objeto con un tipo de objeto, de este modo la invocación sobre una función del tipo se realizara sobre el objeto enlazado. Normalmente esta función es utilizada para llamar a metodos de un tipo base.

%isundefined subsubsection*{isundefined} begin{verbatim}

isundefined(object: Object) -> boolean

end{verbatim} Determina si un objeto no esta definido o asociado a un valor. Retorna un valor de verdad correspondiente.

%isinstance subsubsection*{isinstance} begin{verbatim}

isinstance(object, type | [type...]) -> boolean

end{verbatim} Retorna verdadero si el objeto es una instancia del tipo, si un arreglo de tipos es pasado como segundo argumento el valor de verdad surge de preguntar por cada uno de ellos.

%issubclass subsubsection*{issubclass} begin{verbatim}

issubclass(type1, type2 | [type...]) -> boolean

end{verbatim} Retorna si type1 es una subclase de type2, cuando se pasa un arreglo en lugar de type2 la evaluación se realiza para cada una de las clases incluidas en el mismo.

%Arguments subsubsection*{Arguments} begin{verbatim}

new Arguments(arguments) -> Arguments

end{verbatim} En JavaScript El objeto para los argumentos asociativos debe ir al final de la invocación begin{lstlisting}[style=javascript,label=objeto-arguments,caption=Uniformando argumentos] function unaFuncion(arg1, arg2, arg3) {

var todos = new Arguments(arguments); print('Argumento 1: %s o %s o %s', arg1, todos[0], todos.arg1); print('Argumento 2: %s o %s o %s', arg2, todos[1], todos.arg2); print('Argumento 3: %s o %s o %s', arg3, todos[2], todos.arg3); print('Otros argumentos: %s', todos.args); print('Argumentos pasados por objeto: %o', todos.kwargs);

} function otraFuncion(arg1) {

var todos = new Arguments(arguments, {'def1': 1, 'def2': 2}); print('Argumento 1: %s o %s o %s', arg1, todos[0], todos.arg1); print('Otros argumentos: %s', todos.args); print('Argumentos pasados por objeto: %o', todos.kwargs);

} end{lstlisting}

begin{lstlisting}[style=consola] >>> unaFuncion('uno', 2, null, 3, 4, 5, {'nombre': 'Diego', 'apellido': 'van Haaster'}) Argumento 1: uno o uno o uno Argumento 2: 2 o 2 o 2 Argumento 3: null o null o null Otros argumentos: 3,4,5 Argumentos pasados por objeto: Object nombre=Diego apellido=van Haaster >>> unaFuncion('uno', 2, null, {'nombre': 'Diego', 'apellido': 'van Haaster'}) ... Otros argumentos: Argumentos pasados por objeto: Object nombre=Diego apellido=van Haaster >>> unaFuncion('uno', 2, null, 3, 2, 3, 4) ... Otros argumentos: 3,2,3,4 Argumentos pasados por objeto: Object >>> otraFuncion('uno', 2, {'nombre': 'Diego', 'apellido': 'van Haaster'}) Argumento 1: uno o uno o uno Otros argumentos: 2 Argumentos pasados por objeto: Object def1=1 def2=2 nombre=Diego apellido=van Haaster >>> otraFuncion('uno', 2, {'def1': 'Diego', 'apellido': 'van Haaster'}) Argumento 1: uno o uno o uno Otros argumentos: 2 Argumentos pasados por objeto: Object def1=Diego def2=2 apellido=van Haaster end{lstlisting}

%Template subsubsection*{Template} begin{verbatim}

new Template(destiny, source) -> Template

end{verbatim}

subsubsection*{Dict} begin{verbatim}

new Dict(object) -> Dict

end{verbatim} begin{lstlisting}[style=consola] >>> dic = new Dict({'db': 5, 'template': 2, 'core': 9}) >>> obj = {'un': 'objeto'} >>> dic.set(obj, 10) >>> arreglo = [1,2,3,4,obj] >>> dic.set(arreglo, 50) >>> dic.get('template') 2

>>> dic.get(arreglo) 50 >>> dic.get(obj) 10 >>> dic.items() [[”db”, 5], [”template”, 2], [”core”, 9], [Object un=objeto, 10], [[1, 2, 3, 2 more...], 50]] >>> dic.keys() [”db”, “template”, “core”, Object un=objeto, [1, 2, 3, 2 more...]] >>> dic.values() [5, 2, 9, 10, 50] end{lstlisting}

subsubsection*{Set} begin{verbatim}

new Set(array) -> Set

end{verbatim} Un set es una coleccion de elementos unicos, de forma similar a los conjuntos este objeto soporta intersecciones, uniones, restas, etc. begin{lstlisting}[style=consola] >>> set = new Set([1,2,3,4,5,6,7,8,9,3,6,1,4,7]) >>> len(set) 9 >>> set.add(6) >>> set) 9 >>> set2 = set.intersection([1,3,5,6]) >>> set2.elements [1, 3, 5, 6] end{lstlisting}

%hash subsubsection*{hash} begin{verbatim}

hash(string | number) -> number

end{verbatim} Retorna un valor de hash para el argumento dado, para los mismos argumentos se teronran los mismos valores de hash.

subsubsection*{id} begin{verbatim}

id(value) -> number

end{verbatim} Asigna y retorna un identificador unico para el valor pasado como argumento. Al pasar un valor que sea de tipo objeto la funcion id modificara la estructura interna agregando el atributo verb|__hash__| para ‘‘etiquetar” el objeto y en posteriores llamadas retornara el mismo identificador.

%getattr subsubsection*{getattr} begin{verbatim}

getattr(object, name, default) -> value

end{verbatim} Obtiene un atributo de un objeto mediante su nombre, en caso de pasar un valor por defecto este es retornado si el atributo buscado no esta definido en el objeto, en caso contrario una excepcion es lanzada.

%setattr subsubsection*{setattr} begin{verbatim}

setattr(object, name, value)

end{verbatim} Establece un atributo en un objeto con el nombre pasado. El valor establecido pasa a formar parte del objeto.

%hasattr subsubsection*{hasattr} begin{verbatim}

hasattr(object, name) -> boolean

end{verbatim} Retrona verdadero en caso de que el objeto tenga un atributo con el nombre correspondiente, falso en caso contrario.

%assert subsubsection*{assert} begin{verbatim}

assert(boolean, mesage)

end{verbatim} Chequea que el valor de verdad pasado sea verdadero en caso contrario retorna una excepcion conteniendo el mensaje pasado.

%bool subsubsection*{bool} begin{verbatim}

bool(object)

end{verbatim} Determina el valor de verdad de un objeto pasado, los valores de verdad son como sigue: arreglos, objetos y cadenas vacias en conjunto con los valores null y undefined son falsos; todos los demas casos son verdaderos. En el caso particular de que un objeto defina el metodo verb|__nonzero__| este es invocado para determinar el valor de verdad.

%callable subsubsection*{callable} begin{verbatim}

callable(value) -> boolean

end{verbatim} Retorna verdadero en caso de que el valor pasado sea instancia de una funcion osea pueda ser llamado, falso en caso contrario.

%chr subsubsection*{chr} begin{verbatim}

chr(number) -> character

end{verbatim} Retorna el caracter correspondiente al numero ordinal pasado.

%ord subsubsection*{ord} begin{verbatim}

ord(character) -> number

end{verbatim} Retorna un numero correspondiente al caracter pasado. begin{lstlisting}[style=consola] >>> ord(chr(65)) 65 >>> chr(ord("A")) "A" end{lstlisting}

subsubsection*{bisect} begin{verbatim}

bisect(seq, element) -> position

end{verbatim} Dada una secuencia ordenada y un elemento la funcion bisect retorna un numero referenciando a la posicion en que el elemnto debe ser insertado en la secuencia, para que esta conseve su orden. Si los elementos de la secuencia definen el metodo verb|__cmp__| este es invocado para determinar la posicion a retornar. begin{lstlisting}[style=consola] >>> a = [1,2,3,4,5] >>> bisect(a,6) 5 >>> bisect(a,2) 2 >>> a[bisect(a,3)] = 3 >>> a [1, 2, 3, 3, 5] end{lstlisting}

%equal subsubsection*{equal} begin{verbatim}

equal(object1, object2) -> boolean

end{verbatim} Compara dos objetos determinando el valor de igual para los mismos, verdadero es retornado en caso de ser los dos objetos iguales. En caso de que object1 defina el metodo verb|__eq__| este es invocado con object2 pasado como parametro para determinar la igualdad.

%nequal subsubsection*{nequal} begin{verbatim}

nequal(object, object) -> boolean

end{verbatim} Compara dos objetos determinando el valor de igual para los mismos, verdadero es retornado en caso de ser los dos objetos distintos. En caso de que object1 defina el metodo verb|__ne__| este es invocado con object2 pasado como parametro para determinar la no igualdad.

%number subsubsection*{number} begin{verbatim}

number(object) -> number

end{verbatim} Convierte un objeto a su representacion numerica.

subsubsection*{flatten} begin{verbatim}

flatten(array) -> flattenArray

end{verbatim} Aplana un arreglo de modo que el resultado sea un unico arreglo conteniendo todos los elementos que se pasaron en multiples arreglos a la funcion.

%include subsubsection*{include} begin{verbatim}

include(seq, element) -> boolean

end{verbatim} Determina si un elemento esta incluido en una secuencia o coleccion de objetos, si la coleccion implementa el metodo verb|__contains__|, este es utilizado para determinar la pertenencia del elemento.

%len subsubsection*{len} begin{verbatim}

len(seq) -> boolean

end{verbatim} Retorna un valor numerico representando la cantidad de elementos contenidos en la secuencia o coleccion, si la coleccion implementa el metodo verb|__len__|, este es utilizado para determinar la cantidad de elementos.

subsubsection*{array} begin{verbatim}

array(seq) -> [element...]

end{verbatim} Genera un arreglo en base a la secuencia pasada, si la secuencia implementa el metodo verb|__iter__|, este es utilizado para llenar el arrgelo con los elementos.

subsubsection*{print} begin{verbatim}

print(text...)

end{verbatim} Si la consola de firebug esta instalada este metodo imprime el texto pasado por consola.

subsubsection*{string} begin{verbatim}

string(object)

end{verbatim} Retorna una reprecentacion en texto del objeto pasado como argumento. Si el objeto define el metodo verb|__str__| este es invocado para obtener la reprecentacion.

subsubsection*{values} begin{verbatim}

values(object) -> [value...]

end{verbatim} Retorna un arreglo con los valores del objeto pasado como argumento.

subsubsection*{keys} begin{verbatim}

keys(object) -> [key...]

end{verbatim} Retorna un arreglo con las claves del objeto pasado como argumento.

%items subsubsection*{items} begin{verbatim}

items(object) -> [[key, value]...]

end{verbatim} Retorna en forma de arreglo cada pareja clave, valor de un objeto pasado como argumento. begin{lstlisting}[style=consola] >>> items({'perro': 1, 'gato': 7}) [["perro", 1], ["gato", 7]] end{lstlisting}

subsubsection*{inspect} begin{verbatim}

inspect(object) -> string

end{verbatim}

%unique subsubsection*{unique} begin{verbatim}

unique(array) -> [element...]

end{verbatim} Dado un arreglo con elementos repetidos retorna un nuevo arreglo que se compone de los elementos unicos encontrados.

%range subsubsection*{range} begin{verbatim}

range([begin = 0,] end[, step = 1]) -> [number...]

end{verbatim} Retorna un arreglo conteniendo una progresion aritmetica de numeros enteros. Los parametros son variables y en su invocacion mas simple se pasa solo el final de la secuencia de numeros a generar, asumiendo para ello un inicio en 0 y un incremento en una unidad, estos valores pueden ser modificados. begin{lstlisting}[style=consola] >>> range(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] >>> range(4, 10) [4, 5, 6, 7, 8, 9] >>> range(4, 10, 2) [4, 6, 8] end{lstlisting}

%xrange subsubsection*{xrange} begin{verbatim}

xrange([begin = 0,] end[, step = 1]) -> generator

end{verbatim} Similar a range pero en lugar de retornar un arrglo retorna un objeto que generar los valores bajo demanda.

%zip subsubsection*{zip} begin{verbatim}

zip(seq1 [, seq2 [...]]) -> [[seq1[0], seq2[0] ...], [...]]

end{verbatim} Retorna un arreglo en donde cada secuencia contenida es el resultado de combinar cada una de las secuencias que se pasaron como argumento, la longitud de las secuencias queda acotada a la longitud de la secuencia mas corta. begin{lstlisting}[style=consola] >>> zip([1,2,3,4,5,6], ['a','b','c','d','e','f']) [[1, "a"], [2, "b"], [3, "c"], [4, "d"], [5, "e"], [6, "f"]] >>> zip([1,2,3,4,5,6], ['a','b','c','d','e','f','g','h']) [[1, "a"], [2, "b"], [3, "c"], [4, "d"], [5, "e"], [6, "f"]] >>> zip([1,2,3,4,5,6], ['a','b','c','d']) [[1, "a"], [2, "b"], [3, "c"], [4, "d"], [5, undefined], [6, undefined]] >>> zip([1,2,3,4,5,6], ['a','b','c','d','e','f'], [10,11,12,13,14,15,16]) [[1, "a", 10], [2, "b", 11], [3, "c", 12], [4, "d", 13], [5, "e", 14], [6, "f", 15]] end{lstlisting}

subsection{sys} Este modulo provee acceso a algunos objetos y funciones mantenidas por Protopy y que resultan de utilidad para interactuar con el ambiente. subsubsection*{version} Version de Protopy. subsubsection*{browser} Este objeto provee informacion sobre el navegador en el cual protopy se cargo. begin{itemize}

item{IE} Si el navegador es Internet Explorer. item{Opera} Si el navegador es Opera. item{WebKit} Si el navegador es AppleWebKit. item{Geck} Si el navegador es Gecko. item{MobileSafari} Si el navegador es Apple Mobile Safari. item{fatures} Algunas herramientas que el navegador proveea, por ejemplo

XPath, un selector de css, y otras extensiones. end{itemize}

subsubsection*{gears} Objeto gears para interactuar con el plugin de google gears. installed install factory subsubsection*{register_path} Registra una ruta en el servidor para un paquete, de este modo, las imporaciones de modulos dependientes de de ese paquete se realizaran sobre la ruta asociada. subsubsection*{module_url} Retorna la ruta correspondiente al nombre de modulo pasado. subsubsection*{modules} Un objeto para asociar los nombres de modulos con los modulos propiamente dicho que se van cargando bajo demanda. subsubsection*{paths} Las rutas registradas para la carga de modulos.

subsection{exception} Modulo que reune todas las excepciones que Protopy provee a la hora de mostrar errores, este modulo es cargado en el ambiente cuando Protopy inicia, no siendo necesario su requerimiento posteriormente.

paragraph{Excepciones} Exception, AssertionError, AttributeError, LoadError, KeyError, NotImplementedError, TypeError, ValueError.

subsection{event} Este es el modulo encargado de encapsular la logica de eventos requerida tanto por los elementos del DOM como por el usuario.

paragraph{Funciones} subsubsection*{connect} begin{verbatim}

connect(obj: Object|null, event: String, context: Object|null, method:

String|Function, dontFix: Boolean) -> handle: Handle end{verbatim} Provee un mecanimos para conectar la ejecucion de una funcion a otra o a un evento del DOM.

subsubsection*{disconnect} begin{verbatim}

disconnect(handle: Handle)

end{verbatim} Quita el la relacion establecida por ‘‘connect’‘.

subsubsection*{subscribe} begin{verbatim}

subscribe(topic: String, context: Object|null, method: String|Function) ->

handle: Handle end{verbatim} Subscribe una funcion a un evento de usuario expresado como un texto, cuando el evento ocurra la funcion se ejecuta.

subsubsection*{unsubscribe} begin{verbatim}

unsubscribe(handle: Handle)

end{verbatim} Quita la relacion de la funcion con el evento.

subsubsection*{publish} begin{verbatim}

publish(topic: String, arguments: Array)

end{verbatim} Emite el evento de usuario, provocando la ejecucion de las funciones subscriptas y pasando los argumentos correspondientes.

subsubsection*{connectPublisher} begin{verbatim}

connectPublisher(topic: String, obj: Object, event: String) -> handle: Handle

end{verbatim} Conecta un evento a un evento de usuario asegurando que cada vez que el evento se produsca se llamara a la funcion registrada para el evento de usuario.

subsubsection*{fixEvent} begin{verbatim}

fixEvent(evt: Event, sender: DOMNode)

end{verbatim} Normaliza las propiedades de un evento, tanto pulsasiones del teclado como posiciones x/y del raton.

subsubsection*{stopEvent} begin{verbatim}

stopEvent(evt: Event)

end{verbatim} Detiene el evento, evitando la propagacion y la accion por defecto.

subsubsection*{keys} Objeto que encapsula los codigos de las teclas de funcion y control.

subsection{timer}

paragraph{Funciones} subsubsection*{delay} begin{verbatim}

delay(function)

end{verbatim}

subsubsection*{defer} begin{verbatim}

defer(function)

end{verbatim}

subsection{ajax} Este modulo contiene funcionalidad propia del ajax, para el manejo de peticiones asincronicas al servidor. paragraph{Funciones} subsubsection*{Request} begin{verbatim}

new Request()

end{verbatim} new Request() subsubsection*{Response} begin{verbatim}

new Response()

end{verbatim} new Response() subsubsection*{toQueryParams} begin{verbatim}

toQueryParams(string, separator) -> object

end{verbatim} subsubsection*{toQueryString} begin{verbatim}

toQueryString(params) -> string

end{verbatim}

subsection{dom} Este modulo brinda el soporte para el manejo del DOM de una forma simple para el usuario. paragraph{Funciones} subsubsection*{query} begin{verbatim}

query(cssRule) -> [HTMLElement...]

end{verbatim}

section{Extendiendo Javascript} Protopy no solo aporta modulos y funciones utiles al desarrollo, sino que tambien agrega nueva funcionalidad a los objetos de javascript.

subsection{String} Estos metodos amplian las herramientas para el manejo de cadenas de texto. subsubsection*{sub} begin{verbatim}

    string.sub(pattern, replacement[, count = 1]) -> string

end{verbatim} Retorna una cadena donde la primera ocurrencia del patron pasado es reemplazado por la cadena o cada uno de los valores retornados por la funcion pasada como segundo argumento. El patron puede ser una exprecion regular o una cadena.

subsubsection*{subs} begin{verbatim}

    string.subs(value...) -> string

end{verbatim} Substitulle cada patron encontrado en la cadena por los valores correspondientes, si el primer valor es un objeto, se espera un patron del tipo clave en la cadena para su reemplazo.

subsubsection*{format} begin{verbatim}

    string.format(f) -> string

end{verbatim} Da formato a una cadena de texto, al estilo C.

subsubsection*{inspect} begin{verbatim}

    string.inspect(use_double_quotes) -> string

end{verbatim} Retorna una version de debug de la cadena, esta puede ser con comillas simples o con comillas dobles.

subsubsection*{truncate} begin{verbatim}

    string.truncate([length = 30[, suffix = '...']]) -> string

end{verbatim} Recorta una cadena recortada en la longitud indicada o 30 caracteres por defecto, si se pasa un sufijo este es utilizado para indicar el recorte, sino los ''...'' son utilizados.

subsubsection*{strip} begin{verbatim}

    string.strip() -> string

end{verbatim} Quita los espacios en blanco al principio y al final de una cadena.

subsubsection*{striptags} begin{verbatim}

    string.striptags() -> string

end{verbatim} Quita las etiquetas HTML de una cadena.

subsubsection*{stripscripts} begin{verbatim}

    string.stripscripts() -> string

end{verbatim} Quita todos los bloques ''strips'' de una cadena.

subsubsection*{extractscripts} begin{verbatim}

    string.extractscripts() -> [string...]

end{verbatim} Extrae todos los scripts contenidos en la cadena y los retorna en un arreglo.

subsubsection*{evalscripts} begin{verbatim}

    string.evalscripts() -> [value...]

end{verbatim} Evalua todos los scripts contenidos en la cadena y retorna un arreglos con los resultados de cada evaluacion.

subsubsection*{escapeHTML} begin{verbatim}

string.escapeHTML() -> string

end{verbatim} Convierte los caracteres especiales del HTML a sus entidades equivalentes.

subsubsection*{unescapeHTML} begin{verbatim}

string.unescapeHTML() -> string

end{verbatim} Convierte las entidades de caracteres especiales del HTML a sus respectivos simbolos.

subsubsection*{succ} begin{verbatim}

string.succ() -> string

end{verbatim} Convierte un caracter en el caracter siguiente segun la tabla de caracteres Unicode.

subsubsection*{times} begin{verbatim}

string.times(count[, separator = ‘’]) -> string

end{verbatim} Concatena una cadena tantas veces como se indique, si se pasa un separador, este es utilizado para intercalar.

subsubsection*{camelize} begin{verbatim}

string.camelize() -> string

end{verbatim} Convierte una cadena separada por guiones medios (‘‘-’’) a una nueva cadena tipo ‘’camello’’. Por ejemplo, ‘foo-bar’ pasa a ser ‘fooBar’.

subsubsection*{capitalize} begin{verbatim}

string.capitalize() -> string

end{verbatim} Pasa a mayuscula la primera letra y el resto de la cadena a minusculas.

subsubsection*{underscore} begin{verbatim}

string.underscore() -> string

end{verbatim} Convierte una cadena tipo ‘’camello” a una nueva cadena separada por guiones bajos (‘‘_’’).

subsubsection*{dasherize} begin{verbatim}

string.dasherize() -> string

end{verbatim} Remplaza cada ocurrencia de un guion bajo (`\_”) por un guion medio (-’’`).

subsubsection*{startswith} begin{verbatim}

string.startswith(pattern) -> boolean

end{verbatim} Chequea si la cadena inicia con el patron pasado.

subsubsection*{endswith} begin{verbatim}

string.endswith(pattern) -> boolean

end{verbatim} Chequea si la cadena termina con el patron pasado.

subsubsection*{blank} begin{verbatim}

string.blank() -> boolean

end{verbatim} Chequea si una cadena esta en blanco, esto es si esta vacia o solo contiene espacios en blanco.

subsection{Number} Estos metodos agregan funcionalidad sobre los objetos numericos. subsubsection*{format} begin{verbatim}

number.format(f, radix) -> string

end{verbatim} Da formato a un numero en base a una cadena de texto, al estilo C.

subsection{Date} subsubsection*{toISO8601} begin{verbatim}

date.toISO8601() -> string

end{verbatim} Retorna una representacion de la fecha en ISO8601.

subsection{Element} Extencion sobre los elementos del DOM, simplificando trabajos comunes de desarrollo. subsubsection*{visible} begin{verbatim}

HTMLElement.visible() -> Boolean

end{verbatim} Retorna un valor de verdad que determina si el elemento esta visible al usuario, verificando el atributo de estilo ‘‘display’‘.

subsubsection*{toggle} begin{verbatim}

HTMLElement.toggle() -> HTMLElement

end{verbatim} Alterna la visibilidad del elemento.

subsubsection*{hide} begin{verbatim}

HTMLElement.hide() -> HTMLElement

end{verbatim} Oculta el elemento al usuario, modificando el atributo de estilo.

subsubsection*{show} begin{verbatim}

HTMLElement.show() -> HTMLElement

end{verbatim} Muestra el elemento.

subsubsection*{remove} begin{verbatim}

HTMLElement.remove() -> HTMLElement

end{verbatim} Quita el elemento del documento y lo retorna.

subsubsection*{update} begin{verbatim}

HTMLElement.update(content) -> HTMLElement

end{verbatim} Reemplaza el contendio del elemento con el argumento pasado y terorna el elemento.

subsubsection*{insert} begin{verbatim}

HTMLElement.insert({ position: content }) -> HTMLElement HTMLElement.insert(content) -> HTMLElement

end{verbatim} Inserta conentido al principio, al final, sobre o debajo del elemento, para definir la poscicion de la insercion el argumento se debe pasar en forma de objeto, donde la clave es la pocicion y el valor el contenido a insertar; si el argumento es contenido a insertar este se inserta al final del elemento.

subsubsection*{select} begin{verbatim}

HTMLElement.select(selector) -> HTMLElement

end{verbatim} Toma un numero arbitrario de selectores CSS y retorna un arreglo con los elementos que concuerden con estos y esten dentro del elemento al que se aplica la funcion.

begin{lstlisting}[style=consola] >>> $('PostForm').select('input') [input#id_title, input guardar] >>> $('content').select('div') [div.header, div.breadcrumbs, div.middle, div, div.right, div#clear] >>> $('content').select('div.middle') [div.middle] end{lstlisting}

%Faltan, estoy mal con la forma de los nombres :(CamelCase camel_case % Camelcase camelcase HAAAAAAAAA! %empty: function() %getStyle: function(style) %getOpacity: function(element) %setStyle: function(styles) %setOpacity: function(value)

subsection{Forms} Estos metodos decoran a los elementos del tipo formulario, agregando funcionalidad sobre los mismos y sobre los campos que contienen. subsubsection*{disable} begin{verbatim}

HTMLFormElement.disable() -> HTMLFormElement

end{verbatim} Deshabilita todos los campos de este formulario para el ingreso de valores.

subsubsection*{enable} begin{verbatim}

HTMLFormElement.enable() -> HTMLFormElement

end{verbatim} Habilita todo campos del formulario para el ingreso de valores.

subsubsection*{serialize} begin{verbatim}

HTMLFormElement.serialize() -> object

end{verbatim} Retorna un objeto conteniendo todos campos del formulario serializados con sus respectivos valores.

begin{lstlisting}[style=consola] >>> $('PostForm') <form id="PostForm" method="post" action="/blog/add_post/"> >>> $('PostForm').serialize() Object title=Hola mundo body=Este es un post tags=[1] end{lstlisting}

subsection{Forms.Element} Los metodos que a continuación se presentan decoran a los elementos o campos de un formulario, simplificando y agilizando el trabajo con los mismos. subsubsection*{serialize} begin{verbatim}

HTMLElement.serialize() -> string

end{verbatim} Crea una cadena en URL-encoding representando el contenido del campo expresado como clave=valor, para su uso en una peticion AJAX por ejemplo. Este atributo trabaja sobre un unico campo, si en lugar de esto se requiere serializar todo el formulario vea Form.serialize(). Si se requiere es el valor del campo en lugar de la pareja clave=valor, vea get_value().

subsubsection*{get_value} begin{verbatim}

HTMLElement.get_value() -> value

end{verbatim} Retorna el valor actual del campo. Una cadena de texto es retornada en la mayoria de los casos excepto en el caso de un select multiple, en que se retorna un arreglo con los valores.

subsubsection*{set_value} begin{verbatim}

HTMLElement.set_value(value) -> HTMLElement

end{verbatim} Establese el valor de un campo.

subsubsection*{clear} begin{verbatim}

HTMLElement.clear() -> HTMLElement

end{verbatim} Limpia un campo de texto asignando como valor la cadena vacia.

subsubsection*{present} begin{verbatim}

HTMLElement.present() -> boolean

end{verbatim} Retorna verdadero si el campo de texto tiene un valor asignado, falso en otro caso.

subsubsection*{activate} begin{verbatim}

HTMLElement.activate() -> HTMLElement

end{verbatim} Pone el cursor sobre el campo y selecciona el valor si el campo es del tipo texto.

subsubsection*{disable} begin{verbatim}

HTMLElement.disable() -> HTMLElement

end{verbatim} Deshabilita el campo, impidiendo de este modo que se modifique su valor hasta que sea habilitado nuevamente. Los campos de un formulario que esten deshabilitados no se serializan.

subsubsection*{enable} begin{verbatim}

HTMLElement.enable() -> HTMLElement

end{verbatim} Habilita un campo, previamente deshabilitado, para el ingreso de valores.

section{Otros modulos} subsection{gears} — subsection{logging} — subsection{json} begin{lstlisting}[style=consola] >>> require('json') >>> toJson = {'numero': 1, 'cadena': 'texto', 'arreglo': [1,2,3,4,5,6], 'objeto': {'clave': 'valor'}, 'logico': true} >>> toSend = json.stringify(toJson) "{"numero": 1, "cadena": "texto", "arreglo": [1, 2, 3, 4, 5, 6], "objeto": {"clave": "valor"}, "logico": true}" >>> fromJson = json.parse(toSend) Object numero=1 cadena=texto arreglo=[6] objeto=Object end{lstlisting} subsection{rpc}

begin{lstlisting}[style=python] class MyFuncs:

def _listMethods(self): # this method must be present for system.listMethods # to work return ['add', 'pow']

def _methodHelp(self, method): # this method must be present for system.methodHelp # to work if method == 'add':

return "add(2,3) => 5"

elif method == 'pow': return "pow(x, y[, z]) => number"

else: # By convention, return empty # string if no help is available return ""

def _dispatch(self, method, params): if method == 'pow': return pow(*params)

elif method == 'add': return params[0] + params[1]

else: raise 'bad method'

server = SimpleXMLRPCServer(("localhost", 8000)) dispatcher.register_introspection_functions() dispatcher.register_instance(MyFuncs()) #dispatcher.serve_forever() end{lstlisting}

begin{lstlisting}[style=consola] >>> require('rpc') >>> funcs = new rpc.ServiceProxy('rpc/test', {asynchronous: false}) >>> funcs.system.listMethods() ["add", "pow", "system.listMethods", "system.methodHelp", "system.methodSignature"] >>> for each (m in funcs.system.listMethods())

print(m + ': ' + funcs.system.methodHelp(m));

add: add(2,3) => 5 pow: pow(x, y[, z]) => number system.listMethods: system.listMethods() => ['add', 'subtract', 'multiple']

Returns a list of the methods supported by the server.

system.methodHelp: system.methodHelp('add') => "Adds two integers together" Returns a string containing documentation for the specified method.

system.methodSignature: system.methodSignature('add') => [double, int, int] Returns a list describing the signature of the method. In the above example,

the add method takes two integers as arguments and returns a double result. This server does NOT support system.methodSignature.

```
>>> funcs.add(24,4)
28
>>> funcs.pow(2,4)
16
\end{lstlisting}
```

Add rightarrow verb|{“version”: “1.1”, “method”: “add”, “id”: 1, “params”: [24,4]}| rightarrow verb|{“id”:1,”result”:28}|

Pow rightarrow verb|{“version”: “1.1”, “method”: “pow”, “id”: 2, “params”: [2,4]}| leftarrow verb|{“id”:2,”result”:16}|

Referencia sobre el lenguaje Python

A.1 Modularidad

A.1.1 Ámbito de nombres

TODO

A.1.2 Módulos

Un módulo en Python es un archivo con código python. Usualmente con la extensión .py. Un módulo puede ser importado en el ámbito de nombres local mediante la sentecia **import**.

Por ejemplo, consideremos el módulo funciones.py

```
# coding: utf-8

def media(lista):
        return float(sum(lista)) / len(lista)

def media_geo(lista):
        # La raíz puede expresarse como potencia > 1
        return reduce(lambda x, y: x*y, lista) ** 1.0/len(lista)
```

Para importar el módulo al ámbito de nombres local se puede utilizar la sentencia **import**.

```
>>> import funciones
>>> dir(funciones)
```

A.1.3 Paquete

Un paquete es una colección de uno o más módulos, contenidos en una carpeta.

A.1.4 Módulo de expresiones regulares “re”

El módulo de expresiones regulares de Python permite recuperar gurpos nombrados.

```
r'persona/(?P<nombre>\w+)/(?P<edad>\d{2,3})'
```

En la expresión regular anterior se pueden recuperar el grupo **nombre**, que es un gurpo de uno o más letras, y el grupo **edad**, que es un entero de 2 o 3 cifras.

```
>>> import re
>>> expresion = re.compile(r'persona/(?P<nombre>\w+)/(?P<edad>\d{2,3})')
>>> match = expresion.search('persona/nahuel/25')
>>> match.group('nombre')
'nahuel'
>>> match.group('edad')
'25'
```

A.1.5 Metaprogramación mediante metaclases

Se puede definir la estructura de una clase mediante otra clase que herede de type.

Referencia sobre Django

B.1 Instalación de Django

La mayoría de la gente querrá instalar el lanzamiento oficial más reciente de http://www.djangoproject.com/download/. Django usa el método `distutils` estándar de instalación de Python, que en el mundo de Linux es así:

1. Baja el tarball, que se llamará algo así como *Django-0.96.tar.gz*
2. `tar xzvf Django-*.tar.gz`
3. `cd Django-*`
4. `sudo python setup.py install`

En Windows, recomendamos usar 7-Zip para manejar archivos comprimidos de todo tipo, incluyendo `.tar.gz`. Puedes bajar 7-Zip de http://www.djangoproject.com/r/7zip/.

Cambia a algún otro directorio e inicia `python`. Si todo está funcionando bien, deberías poder importar el módulo `django`:

```
>>> import django
>>> django.VERSION
(0, 96, None)
```

B.2 Comandos del módulo manage

B.2.1 El comando syncdb

El comando syncdb busca los modelos de todas las aplicaciones instaladas. Por cada modelo, genera el SQL necesario para crear las tablas relacionales y mediante la configuración definida en el módulo settings, se connecta con la base de datos y ejecuta las secuencia SQL, creando así las tablas del modelo que no existan.

B.2.2 El comando runserver

Este comando lanza el servidor de desarrollo. Generalmente se ejecuta en el puerto 8000.

B.2.3 El comando validate

Este comando recibe puede no recibir argumentos o una lista de aplicaciones que validar. Realiza una verficación de sintaxis

B.3 Comandos de usuario

Django permite

Índice