The nexml S4 Object

Carl Boettiger Scott Chamberlain Rutger Vos Hilmar Lapp

Understanding the `nexml` S4 object

The `RNeXML` package provides many convenient functions to add and extract information from `nexml` objects in the R environment without requiring the reader to understand the details of the NeXML data structure and making it less likely that a user will generate invalid NeXML syntax that could not be read by other parsers. The `nexml` object we have been using in all of the examples is built on R's S4 mechanism. Advanced users may sometimes prefer to interact with the data structure more directly using R's S4 class mechanism and subsetting methods. Many R users are more familiar with the S3 class mechanism (such as in the `ape` package phylo objects) rather than the S4 class mechanism used in phylogenetics packages such as `ouch` and `phylobase`. The `phylobase` vignette provides an excellent introduction to these data structures. Users already familiar with subsetting lists and other S3 objects in R are likely familar with the use of the `$` operator, such as `phy$edge`. S4 objects simply use an `@` operator instead (but cannot be subset using numeric arguments such as `phy[[1]]` or named arguments such as phy[["edge"]]).

The `nexml` object is an S4 object, as are all of its components (slots). Its hierarchical structure corresponds exactly with the XML tree of a NeXML file, with the single exception that both XML attributes and children are represented as slots.
S4 objects have constructor functions to initialize them. We create a new `nexml` object with the command:

```
nex <- new("nexml")
```

We can see a list of slots contained in this object with

```
slotNames(nex)
```

```
 [1] "version"            "generator"          "xsi:schemaLocation"
 [4] "namespaces"         "otus"               "trees"
 [7] "characters"         "meta"               "about"
[10] "xsi:type"
```

Some of these slots have already been populated for us, for instance, the schema version and default namespaces:

```
nex@version
```

```
[1] "0.9"
```

```
nex@namespaces
```

```
                                        nex
                  "http://www.nexml.org/2009"
                                        xsi
    "http://www.w3.org/2001/XMLSchema-instance"
                                        xml
         "http://www.w3.org/XML/1998/namespace"
                                       cdao
      "http://purl.obolibrary.org/obo/cdao.owl"
```

```
                                          xsd
             "http://www.w3.org/2001/XMLSchema#"
                                           dc
              "http://purl.org/dc/elements/1.1/"
                                       dcterms
                    "http://purl.org/dc/terms/"
                                        prism
"http://prismstandard.org/namespaces/1.2/basic/"
                                           cc
                "http://creativecommons.org/ns#"
                                         ncbi
         "http://www.ncbi.nlm.nih.gov/taxonomy#"
                                           tc
 "http://rs.tdwg.org/ontology/voc/TaxonConcept#"

                    "http://www.nexml.org/2009"
```

Recognize that `nex@namespaces` serves the same role as `get_namespaces` function, but provides direct access to the slot data. For instance, with this syntax we could also overwrite the existing namespaces with `nex@namespaces <- NULL`. Changing the namespace in this way is not advised.

Some slots can contain multiple elements of the same type, such as `trees`, `characters`, and `otus`. For instance, we see that

```
class(nex@characters)
```

```
[1] "ListOfcharacters"
attr(,"package")
[1] "RNeXML"
```

is an object of class `ListOfcharacters`, and is currently empty,

```
length(nex@characters)
```

```
[1] 0
```

In order to assign an object to a slot, it must match the class definition of the slot. We can create a new element of any given class with the `new` function,

```
nex@characters <- new("ListOfcharacters", list(new("characters")))
```

and now we have a length-1 list of character matrices,

```
length(nex@characters)
```

```
[1] 1
```

and we access the first character matrix using the list notation, `[[1]]`. Here we check the class is a `characters` object.

```
class(nex@characters[[1]])
```

```
[1] "characters"
attr(,"package")
[1] "RNeXML"
```

Direct subsetting has two primary use cases: (a) useful in looking up (and possibly editing) a specific value of an element, or (b) when adding metadata annotations to specific elements. Consider the example file

```
f <- system.file("examples", "trees.xml", package="RNeXML")
nex <- nexml_read(f)
```

We can look up the species label of the first `otu` in the first `otus` block:

```
nex@otus[[1]]@otu[[1]]@label
```

```
      label
"species 1"
```

We can add metadata to this particular OTU using this subsetting format

```
nex@otus[[1]]@otu[[1]]@meta <-
  c(meta("skos:note",
         "This species was incorrectly identified"),
        nex@otus[[1]]@otu[[1]]@meta)
```

Here we use the `c` operator to append this element to any existing meta annotations to this otu.