Variations on a . . . theme

Proposals for . . . in R

Doug Kelkhoff @ DSC 2019

2019-09-18

First, an ode to . . .

```
dots_how_do_i_love_thee <- function(...) {
  cat("let me count the ways... \n")
  cat(paste(..., sep = "\n"))
}
```

```
dots_how_do_i_love_thee("really flexible", "mirrors natural language", "intuitive")
```

```
## let me count the ways...
## really flexible
## mirrors natural language
## intuitive
```

The `...` are really handy, but can get unwieldy

- Often need to be collapsed into list, thwarting laziness
- Comes with a cohort of unintuitive operators/functions
- No way to operate on named … elements without evaluation (?)
- Paradigm is only applicable within function calls

```
dots_how_do_i_love_thee <- function(...) {
  cat("let me count the ways... \n")
  cat(paste0("  ", seq(1, ...length()), ". ", list(...), collapse = "\n"))
}

dots_how_do_i_love_thee("really flexible", "mirrors natural language", "intuitive")
```

```
## let me count the ways...
##   1. really flexible
##   2. mirrors natural language
##   3. intuitive
```

Some `...` syntax

`...`, `..n`, `...length()`, `...elt()`

Variadic functions popular in tons of languages

There are plenty of implementations of this feature, some with quite interesting edge case handling

Rosetta Code: Variatic Functions

python

```python
def my_func(*args, **kwargs):
    other_func(*args, **kwargs)
```

julia

```julia
function my_func(args...; kwargs...)
    other_func(args...; kwargs...)
end
```

Ellipsis leverage language familiarity

University of Oxford Style Guide:

- “The quick brown fox jumps over the lazy dog… And if they have not died, they are still alive today.”
- “It is not cold… it is freezing cold.”

Wikipedia “Ellipsis”

They map well to use in verbal languages

- you might expect a list to continue…
- if followed by an ellipsis
- …or a long pause before an end

To generalize…

- `something ...`: more to come
- `... something`: finishing something

How far can we push this familiarity?

Prompt

Are there ways we can extend our intuition for `...` *to other elements of the language?*

Idea 1: Ellipsis unpacking

Composing function calls in R is high bar for new users

```
args <- list("gone!", sep = ", ")
cat(do.call(paste, append(list("going", "going"), args)))
```

- `do.call` assumes pretty strong familiarity of first class functions
- Argument lists must be composed dynamically

Instead, could arguments lists be unpacked directly into a call

```
args <- list("gone!", sep = ", ")
cat(paste("going", "going", ...args))
```

- Retains familiar function call structure
- Syntactically cleaner
- Extends `...` paradigm

Idea 2: Named Ellipsis Parameters

Taking a page from **Julia**, allow naming of a “rest” argument

```
example <- function(dots...) {
  # allow for easier subsetting, manipulation without
  # collapsing to list(...) or handling eval in parent frame
  rainbow_cat(...dots, sep = ", ")
}
```

```
function(rest...) class(rest) # possibly a list of unevaluated promises?
```

Idea 2: Named Ellipsis Parameters

Taking a page from **Julia**, allow naming of a “rest” argument

```
example <- function(dots...) {
  # allow for easier subsetting, manipulation without
  # collapsing to list(...) or handling eval in parent frame

  dots <- dots[!names(dots) %in% "sep"]
  rainbow_cat(...dots, sep = ", ")
}
```

But we still need to handle repeated argument names to avoid `...` induced errors

Idea 3: Better yet, allow repeated arguments

Use ellipses position to indicate precedence

If an argument is passed in ellipses (not explicitly named twice), allow the most recent argument to take precedence.

```
example <- function(...) {
  # fix the 'sep' field regardless of what's in dots
  rainbow_cat(..., sep = ", ")
  #  set a default that is overwritten if present in dots
  rainbow_cat(sep = ", ", ...)
}
```

`julia` *implements ellipses passing as a special case where rightmost argument is used*

Idea 4: Parital Function Application

Appending ellipsis after a function to indicate that it should return a partially applied function instead of the call result

```
newline_cat <- cat(sep = "\n")...
newline_cat("word", "per", "line")
```

```
## word
## per
## line
```

- Retains formals
- Could propegate documentation
- Especially helpful for tab completions

Aside: A mental model for argument unpacking

```
my_function <- function(a, b, c, d, e = 4, dots...) <stuff>
my_args <- list(1, b = 2, c = 3)
my_function(0, a = 2, ...args)
```

Step 1: Consider the function formals

```
# <- what I passed ->       <--- my_function formals -->
  (0, a = 2, ...args)  ==>  (a, b, c, d, e = 4, dots...)
```

Step 2: Fill in formal default values

```
(0, a = 2, 1, b = 2, c = 3, e = 4)
```

Step 3: Align named arguments

```
(a = 2, b = 2, c = 3, e = 4, 0, 1)
```

Step 4: Backfill positional arguments

```
(a = 2, b = 2, c = 3, e = 4, d = 0, dots... = 1)
```

Idea 5: Return list unpacking

Mirror list unpacking into function calls with unpacking into assigned return values

Syntactically parallels function parameter aliasing

```
(x, y, z...) <- list(w = 1, x = 2, y = 3, z = 3)
```

```
> x
## [1] 2

> y
## [1] 3

> z
## $w
## [1] 1
##
## $z
## [1] 3
```

Idea 5: Return list unpacking… considerations

- Can we get rid of the `()`'s?

```
x, y, z... <- list(w = 1, x = 2, y = 3, z = 3)
```

- Requiring unpacking syntax?

```
(x, y, z...) <- ...list(w = 1, x = 2, y = 3, z = 3)
```

- Allowing mapping list names to target object names?

```
(a = x, b = y, c...) <- list(w = 1, x = 2, y = 3, z = 3)
```

- Should the `rest...` contain *just* the remaining values or the entire list?
- Should it be possible to get both?
- Worthwhile having a thunk syntax (_) to throw away list elements?

```
# taking a page from Haskell
# getting both entirety of list (list) and sub-components head (x) & remainder (xs)
f list@(x:xs) = ...
```

Idea 6: Anonymous function shorthand

Draw inspiration from the `purrr` package to create an unambiguous lambda function syntax

```
# function(...) ..1 + ..2
~> ..1 + ..2
```

```
# function(x, y, ...) x + y
x, y ~> x + y
```

- reminescent of `purrr`-style lambda function syntax
- disambiguates lambdas from formulas (`:symbol` shorthand for `name`?)
- currently syntactically invalid, so no breaking changes!

Another alternative for "partial application"

```
new_cat <- ~>cat(sep = ", ", ...)
```

retaining formals and docs require special handling for singular call

Why are these conveniences important to the longevity of the language?

Enter the `Tidyverse`

The tidyverse, and its incredible mindshare, has begun to implement many of these conveniences.

New users have trouble tracking tidyverse-specific syntax

Argument unpacking (and unquoting) `!!!`

```
my_mutations <- list(new_var = "new_var")
mtcars %>% mutate(!!!my_mutations)
```

`purrr`-style lambdas (now in `rlang`) `~`

```
mtcars %>% mutate_at(vars(cyl), ~ . * 2)
```

`ggplot2` symbol representation

```
ggplot(mtcars) +
  aes_(~mpg, ~wt + wt) +  # requires parsing of ~rhs
  geom_point()
```

Reconciling the Tidyverse

Some of the proposed syntax can be used to bring consistency to the tidyverse/base bifurcation

Argument unpacking

```
my_mutations <- list(new_var = "new_var")
mtcars %>% mutate(...my_mutations)
```

handles unpacking, but not unquoting

Lambdas

```
mtcars %>% mutate_at(vars(cyl), x ~> x * 2)
```

Name Notation

```
ggplot(mtcars) +
  aes(:mpg, :wt + :wt) +
  geom_point()
```

Closing Thoughts

```
dots_how_do_i_love_thee <- function(dots...) {
  class(dots)                              # - list of unevaluateded promise?
  names(dots)                              # - operate on list without evaluating
  (pdots, ndots) <- ...split_named(dots)   # - define helpers that keep laxiness
  cat("let me count the ways... \n")
  cat(paste0("  ", seq_along(pdots), ". ", pdots, collapse = "\n", ...ndots, sep = " "))
}
```

- `...` is awesome syntactic feature in R, balancing usability against readability
- Offers opportunities for expanding on paradigm
- Developers benefit from handling `rest...` args without breaking laziness
- Users benefit from consistency of `...` arguments
- Consistency among package implementations reduces bucketing of expectations (e.g. `tidyverse` vs `base`)

Questions & Discussion

Special Thanks

Michael Lawrence, Gabe Becker

Genentech, Roche