

# Working with acs.R (version 2.0)

Ezra Haber Glenn, AICP  
Massachusetts Institute of Technology  
Department of Urban Studies & Planning

March 8, 2016

## Contents

<b>1 Purpose...</b>	<b>2</b>
1.1 of this Document . . . . .	2
1.2 of this Package . . . . .	2
<b>2 Getting Started in R</b>	<b>2</b>
2.1 Getting and Installing R . . . . .	2
2.2 Getting and Installing the <code>acs</code> Package . . . . .	3
2.2.1 Installing from CRAN . . . . .	3
2.2.2 Installing from a zipped tarball . . . . .	4
2.3 Getting and Installing a Census API Key . . . . .	5
2.3.1 Using a Blank Key: an informal workaround . . . . .	6
<b>3 Working with the New Functions</b>	<b>6</b>
3.1 Overview . . . . .	6
3.2 User-Specific Geographies . . . . .	6
3.2.1 Basic Building Blocks: the single element <code>geo.set</code> . . . . .	6
3.2.2 But where's the data...? . . . . .	8
3.2.3 Real <code>geo.sets</code> : complex groups and combinations . . . . .	8
3.2.4 Changing <code>combine</code> and <code>combine.term</code> . . . . .	10
3.2.5 Nested and Flat <code>geo.sets</code> . . . . .	11
3.2.6 Subsetting <code>geo.sets</code> . . . . .	12
3.2.7 Two tools to reduce frustration in selecting geographies . . . . .	12
3.3 Getting Data . . . . .	16
3.3.1 <code>acs.fetch()</code> : the workhorse function . . . . .	16
3.3.2 More descriptive variable names: <code>col.names=</code> . . . . .	18
3.3.3 the <code>acs.lookup()</code> function: finding the variables you want . . . . .	20
<b>4 Exporting Data</b>	<b>25</b>
<b>5 Additional Resources</b>	<b>26</b>

<b>A A worked example using blockgroup-level data and nested combined geo.sets</b>	<b>27</b>
A.1 Making the <code>geo.set</code>	27
A.2 Using <code>combine=T</code> to make a neighborhood	28
A.3 Even more complex <code>geo.sets</code>	29
A.4 Gathering neighborhood data on transit mode-share	30

## 1 Purpose...

### 1.1 of this Document

These notes are intended to accompany the updated `acs.R` package (version 2.0). Much of the code in the package, as well as the guidance in this document, was originally developed by the author in partnership with and under contract for Puget Sound Regional Council (PSRC).<sup>1</sup>

### 1.2 of this Package

The `acs.R` package includes functions to allow users (a) to create custom geographies by combining existing ones provided by the Census, (b) to download and import demographic data from the American Community Survey (ACS) and Decennial Census (SF1/SF3), and (c) manage, manipulate, analyze, and present this data (including proper statistical techniques for dealing with estimates and standard errors). In addition, the package includes a pair of helpful “lookup” tools, one to help users identify the geographic units they want, and the other to identify tables and variables from the ACS for the data they are looking for, and some additional convenience functions for working with Census data.

## 2 Getting Started in R

### 2.1 Getting and Installing R

R is a complete statistical package—actually, a complete programming language with special features for statistical applications—with a syntax and workflow all its own. Luckily, it is well-documented through a variety of tutorials and manuals, most notably those hosted by the `cran` project at <http://cran.r-project.org/manuals.html>. Good starting points include:

- *R Installation and Administration*, to get you started (with chapters for each major operating system); and
- *An Introduction to R*, which provides an introduction to the language and how to use R for doing statistical analysis and graphics.

<sup>1</sup>For users of previous versions of the package, see [http://papers.ssrn.com/sol3/papers.cfm?abstract\\_id=2552524](http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2552524), which provides an earlier version of this guide.

Beyond these, there are dozens of additional good guides. (For a small sampling, see [cran.r-project.org/other-docs.html](http://cran.r-project.org/other-docs.html).)

Exact installation instructions vary from one operating system or distribution to the next, but at this point most include an automated installer of one kind or another (a windows `.exe` installer, a Macintosh `.pkg`, a Debian `apt` package, etc.). Once you have the correct version to install, it usually requires little more than double-clicking an installer icon or executing a single command-line function.

Windows users may also want to review the FAQ at <http://cran.r-project.org/bin/windows/base/rw-FAQ.html>; similarly, Mac users should visit <http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html>.

## 2.2 Getting and Installing the `acs` Package

### 2.2.1 Installing from CRAN

The `acs` package is hosted on the CRAN repository. Once R is installed and started, users may install the package with the `install.packages` command, which automatically handles dependencies.

```
> # do this once, you never need to do it again
# you may be asked to select a CRAN mirror, and then
# lots of output will scroll past
> install.packages("acs")
--- Please select a CRAN mirror for use in this session ---
Loading Tcl/Tk interface ... done
trying URL 'http://lib.stat.cmu.edu/R/CRAN/src/contrib/acs_1.0.tar.gz'
Content type 'application/x-gzip' length 1437111 bytes (1.4 Mb)
opened URL
=====
downloaded 1.4 Mb

* installing *source* package acs ...
** package acs successfully unpacked and MD5 sums checked
** R
** data
** moving datasets to lazyload DB
** inst
** preparing package for lazy loading
Creating a generic function for summary from package base in
  package acs
Creating a new generic function for apply in package acs
Creating a generic function for plot from package graphics in
  package acs
** help
*** installing help indices
** building package indices
```

```
** testing if installed package can be loaded
```

```
* DONE (acs)
```

```
The downloaded source packages are in  
  /tmp/RtmppeCyGj/downloaded_packages
```

```
>
```

```
> # once installed, be sure to load the library:
```

```
> library(acs)
```

After installing, be sure to load the package with `library(acs)` each time you start a new session.

## 2.2.2 Installing from a zipped tarball

If for some reason the latest version of the package is not available through the CRAN repository (or if, perhaps, you intend to experiment with additional modifications to the source code), you may obtain the software as a “zipped tarball” of the complete package. It can be installed just like any other package, although dependencies must be managed separately. Simply start R and then type:

```
> # do this once, you never need to do it again  
> install.packages(pkgs = "acs_2.0.tar.gz", repos = NULL)  
* installing *source* package acs ...  
** R  
** data  
** moving datasets to lazyload DB  
** inst  
** preparing package for lazy loading  
Creating a generic function for summary from package base in  
  package acs  
Creating a new generic function for apply in package acs  
Creating a generic function for plot from package graphics in  
  package acs  
** help  
*** installing help indices  
** building package indices  
** testing if installed package can be loaded  
  
* DONE (acs)  
  
>
```

(You may need to change the working directory to find the file, or specify a complete path to the `pkgs =` argument.) Once installed, don't forget to actually *load* the package to make the installed functions available:

```
> # do this every time to start a new session
> library(acs)
Loading required package: stringr
Loading required package: plyr
Loading required package: XML
```

Attaching package: acs

The following object(s) are masked from package:base:

```
apply
```

```
>
```

The acs.R package depends on a few other fairly common R packages: `methods`, `stringr`, `plyr`, and `XML`. If these are not already on your system, you may need to install those as well—just use `install.packages("package.name")`. (Note: when the package is downloaded from the CRAN repository, these dependencies will be managed automatically.)

If installation of the tarball fails, users may need to specify the following additional options (likely for Windows and possibly Mac systems):

```
> install.packages("/path/to/acs_1.0.tar.gz", repos = NULL,
  type = "source")
```

Assuming you were able to do these steps, we're ready to try it out.

### 2.3 Getting and Installing a Census API Key

To download data via the American Community Survey application program interface (API), users need to request a “key” from the Census. Visit [http://api.census.gov/data/key\\_signup.html](http://api.census.gov/data/key_signup.html) and fill out the simple form there, agree to the Terms of Service, and the Census will email you a secret key for only you to use.

When working with the functions described below,<sup>2</sup> this key must be provided as an argument to the function. Rather than expecting you to provide this long key each time, the package includes an `api.key.install()` function, which will take the key and install it on the system as part of the package for all future sessions.

```
> # do this once, you never need to do it again
> api.key.install(key="592bc14cnotarealkey686552b17fda3c89dd389")
>
```

---

<sup>2</sup>or at least those that require interaction with the API, such as `acs.fetch()`, `acs.lookup()`, and the `check=` option for `geo.make()`.

### 2.3.1 Using a Blank Key: an informal workaround

Currently, the requirement for a key seems to be laxly enforced by the Census API, but is nonetheless coded into the `acs` package. Users without a key may find success by simply installing a blank key (i.e., via `api.key.install(key="");` similarly, calls to `acs.fetch` and `geo.make(..., check=T)` may succeed with a `key=""` argument. Note that while this may work today, it may fail in the future if the API decides to start enforcing the requirement.

## 3 Working with the New Functions

### 3.1 Overview

We've tried to make this User Guide as detailed as possible, to help you learn about the many advanced features of the new package. As a result, it may look like there is a lot to learn, but in fact the basics are pretty simple: to get ACS data for your own user-defined geographies, all you need to do is:

1. install and load the package, and (optionally) install an API key (see sections 2.2 and 2.3);
2. create a `geo.set` using the `geo.make()` function (see section 3.2);
3. optionally, use the `acs.lookup()` function to explore the variables you may want to download (see section 3.3.3 on page 20) ;
4. use the `acs.fetch()` function to download data for your new geography (see section 3.3.1 on page 16); and then
5. use the existing functions in the package to work with your data (see worked example in appendix A and the package documentation).

As a teaser, here you can see one single command that will download ACS data on “Place of Birth for the Foreign-Born Population in the United States” for *every tract* in all four PSRC counties:

```
> lots.o.data=acs.fetch(geo=geo.make(state="WA",
  county=c(33,35,53,61), tract="*"), endyear=2014, table.number="B05006")
```

When I tried this at home, it took about 10 seconds to download—but it's a lot of data to deal with: over 249,000 numbers (estimates and errors for 161 variables for each of a 776 tracts...).

### 3.2 User-Specific Geographies

#### 3.2.1 Basic Building Blocks: the single element `geo.set`

The `geo.make()` function is used to create new (user-specified) geographies. At the most basic level, a user specifies some combination of existing census levels

(state, county, county subdivision, place, tract, and/or block group), and the function returns a new `geo.set` object holding this information.<sup>3</sup> If you assign this object to a name, you can keep it for later use. (Remember, by default, functions in R don't save things—they simply evaluate and print the results and move on.)

```
> washington=geo.make(state=53)
> alabama=geo.make(state="Alab")
> yakima=geo.make(state="WA", county="Yakima")
> yakima
An object of class "geo.set"
Slot "geo.list":
[[1]]
"geo" object: [1] "Yakima County, Washington"

Slot "combine":
[1] FALSE

Slot "combine.term":
[1] "aggregate"
```

When specifying the state, county, county subdivision, and/or place, `geo.make()` will accept either FIPS codes or common names, and will try to match on partial strings; there is also limited support for regular expressions, but by default the searches are case sensitive and matches are expected at the start of names. (For example, `geo.make(state="WA", county="Kits")` should find Kitsap County, and the more adventurous `yakima=geo.make(state="Washi", county=".*kima")` should work to create the same Yakima county `geo.set` as above.) Important: when creating new geographies, each set of arguments must match with *exactly one* known Census geography: if, for example, the names of two places (or counties, or whatever) would both match, the `geo.make()` function will return an error.<sup>4</sup> The one exception to this “single match” rule is that for the *smallest* level of geography specified, a user can enter "\*" to indicate that *all* geographies at that level should be selected.

`tract=` and `block.group=` can only be specified by FIPS code number (or "\*" for all); they don't really have names to use. (Tracts should be specified as *six digit* numbers, although initial zeroes may be removed; often *trailing* zeroes

---

<sup>3</sup>Note: for reasons that will become clear in a moment, even a single geographic unit—say, one specific tract or county—will be wrapped up as a `geo.set`. Technically, each individual *element* in the set is known as a `geo`, but users will rarely (if ever) interact with individual elements such as this; wrapping all groups of geographies—even groups consisting of just one element—in `geo.sets` like this will help make them easier to deal with as the geographies get more complex. To avoid extra words here, I may occasionally ignore this distinction and refer to user-created `geo.sets` as “geos.”

<sup>4</sup>This seemed preferable to simply including both matches, since all sorts of place names might match a string, and it is doubtful a user really wants them all.

are removed in common usage, so a tract referred to as “tract 243” is technically FIPS code 24300, and “tract 3872.01” becomes 387201.)

When creating new geographies, note, too, that not all combinations are valid;<sup>5</sup> in particular, the package attempts to follow paths through the Census “summary levels” (such as summary level 140: “state-county-tract” or summary level 160: “state-place”). So when specifying, for example, state, county, and place, the county will be ignored.

```
> moxee=geo.make(state="WA", county="Yakima", place="Moxee")
Warning message:
In function (state, county, county.subdivision, place, tract, block.group) :
  Using sumlev 160 (state-place)
  Other levels not supported by census api at this time
```

(Despite this warning, the `geo.set` named `moxee` was nonetheless created—this is just a warning.)

### 3.2.2 But where’s the data...?

Note that these new `geo.sets` are simply placeholders for geographic entities—they do not actually contain any census data *about* these places. Be patient (or jump ahead to section 3.3 on page 16).

### 3.2.3 Real `geo.sets`: complex groups and combinations

OK, so far, so good, but what if we want to create new complex geographies made of more than one known census geography? This is why these things are called `geo.sets`: they are actually *collections* of individual census geographic units, which we will later use to download and manipulate ACS data.

Looking back to when we created the `yakima geo.set` object (section 3.2.1 on the preceding page), you can see that the newly created object contained some additional information beyond the name of the place: in particular, all `geo.sets` include a slot named `"combine"` (initially set to `FALSE`) and a slot named `"combine.term"` (initially set to `"aggregate"`). When a `geo.set` consists of just a single `geo`, these extra slots don’t do much, but if a `geo.set` contains more than one item, these two variables determine whether the geographies are to be treated as a set of individual lines or combined together (and relabeled with the `"combine.term"`).<sup>6</sup> Once we have some more interesting sets, these will come in handy.

To make some more interesting sets, we have a few different options:

**Specifying Multiple Geographies through `geo.make()`** Rather than specifying a single set of FIPS codes or names, a user can pass the `geo.make()`

---

<sup>5</sup>But don’t fret: see section 3.2.7 on page 12.

<sup>6</sup>All this combining and relabeling takes place when the actual data is downloaded, so up until then you can continue to change and re-change the structure of your `geo.sets`.



function *vectors* of any length for `state=`, `county=`, and the like. If these vectors are all the same length, they will be combined in sequence; if some are shorter, they will be “recycled” in standard R fashion. (Note that this means if you only specify one item for say, `state=`, it will be used for all, but if you give two states, they will be alternated in the matching.) For simple combinations, this is probably the easiest way to create sets, but for more complicated things, it can get confusing.

```
> psrc=geo.make(state="WA", county=c(33,35,53,61))
> psrc
An object of class "geo.set"
Slot "geo.list":
[[1]]
"geo" object: [1] "King County, Washington"

[[2]]
"geo" object: [1] "Kitsap County, Washington"

[[3]]
"geo" object: [1] "Pierce County, Washington"

[[4]]
"geo" object: [1] "Snohomish County, Washington"

Slot "combine":
[1] FALSE

Slot "combine.term":
[1] "aggregate"
```

**Adding Existing geo.sets with "+"** If you have already created a few different `geo.sets`, you can easily combine them together into a new `geo.set` with the "+" operator. Note that this will create a “flat” `geo.set` (no nesting—see section 3.2.5 on page 11), regardless of whether the constituent parts are nested sets.<sup>7</sup>

```
> north.mercer.island=geo.make(state=53, county=33,
    tract=c(24300,24400))
> optional.tract=geo.make(state=53, county=33, tract=24500)
> # add in one more tract to create new, larger geo
> north.mercer.island.plus=north.mercer.island +
    optional.tract
```

---

<sup>7</sup>By default, the new set will have `combine=FALSE`, with one exception: when adding a single-geography (i.e., `length==1`) to an existing set with `combine=` already set to `TRUE`, the new set will keep `combine=TRUE`, essentially “folding in” the new geography.

```

> length(north.mercer.island.plus)
[1] 3
> str(north.mercer.island.plus)
Formal class 'geo.set' [package "acs"] with 3 slots
 ..@ geo.list      :List of 3
 .. ..$ :Formal class 'geo' [package "acs"] with 3 slots
 .. . . . .@ api.for:List of 1
 .. . . . . . $ tract: num 24300
 .. . . . .@ api.in :List of 2
 .. . . . . . $ state : num 53
 .. . . . . . $ county: num 33
 .. . . . .@ name   : chr "Tract 24300, King County, Washington"
 .. ..$ :Formal class 'geo' [package "acs"] with 3 slots
 .. . . . .@ api.for:List of 1
 .. . . . . . $ tract: num 24400
 .. . . . .@ api.in :List of 2
 .. . . . . . $ state : num 53
 .. . . . . . $ county: num 33
 .. . . . .@ name   : chr "Tract 24400, King County, Washington"
 .. ..$ :Formal class 'geo' [package "acs"] with 3 slots
 .. . . . .@ api.for:List of 1
 .. . . . . . $ tract: num 24500
 .. . . . .@ api.in :List of 2
 .. . . . . . $ state : num 53
 .. . . . . . $ county: num 33
 .. . . . .@ name   : chr "Tract 24500, King County, Washington"
 ..@ combine      : logi FALSE
 ..@ combine.term : chr "aggregate + aggregate"
>

```

**Combining geo.sets with "c()"** A third way to create new multi-element geo.sets is through the use of R's `c()` function (short for “combine”). Similar to the way R treats lists with this function, `c()` will combine geo.sets, but attempt to keep whatever structure they already have in place. The result is often a much more complex kind of nested object. There is real power in this structure, but it can also be a bit tricky; probably best reserved for “power users,” but certainly worth playing with. (Hint: try creating different sets and combining them in different ways with `c()`, and then using `length()` and `str()` to examine the results.)

### 3.2.4 Changing combine and combine.term

To check the current value of the `combine` and `combine.term` slots, you can use the `combine()` and `combine.term()` functions; to change these values, simply use `combine()=` and `combine.term=`.<sup>8</sup>

<sup>8</sup>or `combine()<-` and `combine.term()<-`, for R traditionalists...

```

> combine(north.mercer.island)
[1] FALSE
> combine.term(north.mercer.island)
[1] "aggregate"
> combine(north.mercer.island)=T
> combine.term(north.mercer.island)="North Mercer Island"
> north.mercer.island
An object of class "geo.set"
Slot "geo.list":
[[1]]
"geo" object: [1] "Tract 24300, King County, Washington"

[[2]]
"geo" object: [1] "Tract 24400, King County, Washington"

Slot "combine":
[1] TRUE

Slot "combine.term":
[1] "North Mercer Island"

```

### 3.2.5 Nested and Flat geo.sets

Remember: by default, the addition operator (" $+$ ") will always return “flat” geo.sets, with all the geographies in a single list. The combination operator (" $c()$ "), on the other hand, will generally return nested hierarchies, embedding sets within sets. When working with nested sets like this, the `combine` flag can be set at each level to aggregate *subsets* within the structure (although be careful—if a higher level of set includes `combine=T`, you’ll never actually see the unaggregated subsets deeper down...).

Using these different techniques, you should be able to create whatever sort of new geographies you want—aggregating some geographies, keeping others distinct (but still bundled as a “set” for convenience), mixing and matching different levels of Census geography, and so on.

Two more helpful shortcuts to keep this all straight:

**Setting `combine=` when creating geo.sets** When creating new user-defined geographies with `geo.make()`, a user can explicitly set both `combine=new-value` and `combine.term=new-value` as additional arguments to the function.

`flatten.geo.set()` The package also includes a `flatten.geo.set()` helper function which will iron out even the most complex nested geo.set; it will always return an un-nested `geo.set` with all the geographies at a single depth, with a `length()` equal to the number of composite parts.

### 3.2.6 Subsetting geo.sets

Sometimes, instead of combing geo.sets, users may want to work with just a portion of the an existing set. For this, rather than extending the addition metaphor and developing some sort of “subtraction rule,” the package implements methods for R’s standard subsetting rules for vectors, using [square brackets].

```
> north.mercer.island[2]
An object of class "geo.set"
Slot "geo.list":
[[1]]
"geo" object: [1] "Tract 24400, King County, Washington"
```

```
Slot "combine":
[1] FALSE
```

```
Slot "combine.term":
[1] "aggregate (partial)"
```

```
> psrc[3:4]
An object of class "geo.set"
Slot "geo.list":
[[1]]
"geo" object: [1] "Pierce County, Washington"
```

```
[[2]]
"geo" object: [1] "Snohomish County, Washington"
```

```
Slot "combine":
[1] FALSE
```

```
Slot "combine.term":
[1] "aggregate (partial)"
```

```
>
```

Note that subsetting geo.sets will still always return a complete geo.set, even when selecting only a single geography.

### 3.2.7 Two tools to reduce frustration in selecting geographies

**geo.lookup(): a helper to find what you need** It can often be difficult to find exactly the geography you are looking for, and since (as noted above) **geo.make()** expects single matches to the groups of arguments it is given, this

could result in a lot of frustration—especially when trying to find names for places or county subdivisions, which are unfamiliar to many users (and often seem very close or redundant: e.g., knowing whether to look for “Moses Lake city” vs. “Moses Lake CDP”). To help, the package also includes the `geo.lookup()` function, which searches on the same arguments as `geo.make()`, but outputs *all* the matches for your inspection.

Unlike `geo.make()`, `geo.lookup()` looks for matches *anywhere* in the name (except when dealing with state names), and will output a dataframe showing candidates that match some or all of the arguments. (The logic is a little complicated, but basically to be included, a geography *must* match the given state name; when a county and a subdivision are both given, both must match; otherwise, geographies are included that match any—but not necessarily all—of the other arguments.)

```
> geo.lookup(state="WA", county="Ska", county.subdivision="oo")
  state state.name county      county.name county.subdivision
1    53 Washington   NA          <NA>             NA
2    53 Washington   57 Skagit County             NA
3    53 Washington   59 Skamania County             NA
4    53 Washington   57 Skagit County             92944
5    53 Washington   59 Skamania County             90424
  county.subdivision.name
1                <NA>
2                <NA>
3                <NA>
4      Sedro-Woolley CCD
5      Carson-Underwood CCD
>
> geo.lookup(state="WA", county="Kit", place="Ra")
  state state.name county      county.name place      place.name
1    53 Washington   NA          <NA>     NA          <NA>
2    53 Washington   35 Kitsap County     NA          <NA>
3    53 Washington   37 Kittitas County     NA          <NA>
4    53 Washington   NA Pierce County 57140 Raft Island CDP
5    53 Washington   NA Thurston County 57220 Rainier city
6    53 Washington   NA King County 57395 Ravensdale CDP
7    53 Washington   NA Pacific County 57430 Raymond city
>
```

In the first example, the first row matches just the state (summary level 40); the next two rows show matches at the state and county level (summary level 50); the final two rows show matches that were found looking at state (“WA”), county (containing “Ska”), *and* county subdivision (containing “oo”). In the second example, we see something similar in the first three rows, but after that the rest only match on state-place, ignoring the county selection (like summary level 160), although the county names are included in the output for convenience.

The `geo.lookup()` function can also accept more than a single string for each argument. In the case of states, the function checks each one independently; in all other cases, matching is done on any and all together (as with a logical “or”).<sup>9</sup>

```
> geo.lookup(state=c("WA", "OR"), county=c("M","B"))
  state state.name county      county.name
1    53 Washington   NA             <NA>
2    53 Washington    5    Benton County
3    53 Washington   45    Mason County
4    41    Oregon    NA             <NA>
5    41    Oregon     1    Baker County
6    41    Oregon     3    Benton County
7    41    Oregon   45    Malheur County
8    41    Oregon   47    Marion County
9    41    Oregon   49    Morrow County
10   41    Oregon   51 Multnomah County
>
```

**Setting `check=T` when using `geo.make()`** Another trick to ensure valid geography matching is to set the `check=` argument when using `geo.make()`. When this option is set to `TRUE` (*not* the default), R will verify each element of the `geo.set` in turn as it creates it, querying the Census API server. If it encounters an invalid geography, the function will return an error, saving you trouble later; essentially, it helps catch geographies that are technically valid in form but match to no actual census geographies.<sup>10</sup>

```
> no.state=geo.make(state=3) # there is no state with this FIPS code
An object of class "geo.set"
Slot "geo.list":
[[1]]
"geo" object: character(0)

Slot "combine":
[1] FALSE

Slot "combine.term":
[1] "aggregate"

> no.state-geo.make(state=3, check=T)
```

---

<sup>9</sup>At present, `geo.lookup()` only accepts and searches on `state=`, `county=`, `county.subdivision=`, and `place=`; eventually we hope to include lookup support to help find tract and block.group numbers as well.

<sup>10</sup>At present, the function breaks on the first non-match, without a whole lot of help; in the future I'll add in some better error-handling for this.

```

Testing geography item 1: .... Error in file(file, "rt") : cannot
open the connection

> # give it something with a bad county/tract match
> shoreline.nw.border=geo.make(state=53,
  county=c(33, 33, 61, 61, 61),
  tract=c(20100, 20200, 20300, 50600, 50700), check=T, combine=T,
  combine.term="Shoreline NW Tracts")
Testing geography item 1: Tract 20100, King County, Washington .... OK.
Testing geography item 2: Tract 20200, King County, Washington .... OK.
Testing geography item 3: Tract 20300, Snohomish County, Washington
.... Error in file(file, "rt") : cannot open the connection
>
> # fix the problem and try again
> shoreline.nw.border=geo.make(state=53,
  county=c(33, 33, 33, 61, 61),
  tract=c(20100, 20200, 20300, 50600, 50700), check=T, combine=T,
  combine.term="Shoreline NW Tracts")
Testing geography item 1: Tract 20100, King County, Washington .... OK.
Testing geography item 2: Tract 20200, King County, Washington .... OK.
Testing geography item 3: Tract 20300, King County, Washington .... OK.
Testing geography item 4: Tract 50600, Snohomish County, Washington .... OK.
Testing geography item 5: Tract 50700, Snohomish County, Washington .... OK.
> shoreline.nw.border
An object of class "geo.set"
Slot "geo.list":
[[1]]
"geo" object: [1] "Tract 20100, King County, Washington"

[[2]]
"geo" object: [1] "Tract 20200, King County, Washington"

[[3]]
"geo" object: [1] "Tract 20300, King County, Washington"

[[4]]
"geo" object: [1] "Tract 50600, Snohomish County, Washington"

[[5]]
"geo" object: [1] "Tract 50700, Snohomish County, Washington"

Slot "combine":
[1] TRUE

Slot "combine.term":

```

```
[1] "Shoreline NW Tracts"
```

```
> # it worked!  
> # also, note how we can set combine= and combine.term=  
> # as arguments to geo.make() -- cool!
```

### 3.3 Getting Data

Once you've created some `geo.sets`, you're ready for the fun part: using the package to download data directly from the Census ACS API.<sup>11</sup>

#### 3.3.1 `acs.fetch()`: the workhorse function

Whereas the previous version of the package required users to download data from the Census and then import it into R via the `read.acs()` function, these steps are combined in the new `acs.fetch()` function. Assuming you've already installed an API key (see section 2.3 on page 5)<sup>12</sup>, the call is quite simple:

```
> # table B01003: "Total Population"  
> acs.fetch(geography=psrc, endyear=2011, table.number="B01003")  
ACS DATA:  
2007 -- 2011 ;  
  Estimates w/90% confidence intervals;  
  for different intervals, see confint()  
      B01003_001  
King County      1908379 +/- 0  
Kitsap County    249238 +/- 0  
Pierce County    791528 +/- 0  
Snohomish County 704536 +/- 0  
  
> # table B05001: "Nativity and Citizenship Status in the United States"  
> acs.fetch(geography=north.mercer.island.plus, endyear=2011,  
  table.number="B05001")  
ACS DATA:  
2007 -- 2011 ;  
  Estimates w/90% confidence intervals;  
  for different intervals, see confint()  
      B05001_001  B05001_002  B05001_003  B05001_004  B05001_005  
Census Tract 243 6771 +/- 374 5233 +/- 431 0 +/- 92  71 +/- 74  896 +/- 225  
Census Tract 244 3040 +/- 253 2272 +/- 266 13 +/- 21  57 +/- 44  311 +/- 91  
Census Tract 245 4630 +/- 245 3878 +/- 228 0 +/- 92  69 +/- 43  483 +/- 137  
      B05001_006
```

<sup>11</sup>Actually, you could download data even without creating a `geo.set` object first—R's evaluation procedures are perfectly happy letting you use `geo.make()` “on the fly” and passing the results to the `acs.fetch()` function: you could enter something like `acs.fetch(geography=geo.make(state="WA", county="*"), table.number="B01003")`.

<sup>12</sup>And if you haven't, you can simply add a `key=` argument each time.



```
Census Tract 243 571 +/- 177
Census Tract 244 387 +/- 140
Census Tract 245 200 +/- 85
>
```

For each of these geo.sets, combine=F, but if we want to get more creative we can try:

```
> combine(north.mercer.island.plus)=T
> combine.term(north.mercer.island.plus)="North Mercer Island Tracts"
> my.geos=c(psrc, north.mercer.island.plus, shoreline.nw.border)
> # table B08013: "Aggregate Travel Time To Work (in Minutes) Of Workers By Sex"
> acs.fetch(geo=my.geos, table.number="B08013", endyear=2011,
           col.names=c("Total", "Male", "Female"))
```

ACS DATA:

```
2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
  Total
King County          24971250 +/- 189173
Kitsap County        3183505 +/- 83983
Pierce County        9986285 +/- 116148
Snohomish County     9638070 +/- 109605
North Mercer Island Tracts 118285 +/- 10711.36657948
Shoreline NW Tracts  283540 +/- 19482.3119007986
  Male
King County          13972415 +/- 124050
Kitsap County        1936155 +/- 70636
Pierce County        5787210 +/- 82246
Snohomish County     5550680 +/- 77512
North Mercer Island Tracts 70055 +/- 8217.77110900517
Shoreline NW Tracts  158090 +/- 15076.8047012621
  Female
King County          10998835 +/- 129473
Kitsap County        1247345 +/- 50974
Pierce County        4199075 +/- 77010
Snohomish County     4087390 +/- 67221
North Mercer Island Tracts 48235 +/- 6455.83534486436
Shoreline NW Tracts  125450 +/- 11813.2511189765
```

As you can see, when combine=T, acs.fetch will aggregate the data (using the sum method for acs-class objects) when it is downloaded.<sup>13</sup>

<sup>13</sup>Note: At the request of some users, the acs package includes a special one.zero= option for the sum function, which may be desirable when aggregating lots of variables with zero-values for estimates. Since acs.fetch calls sum internally, you can set this option when you call acs.fetch and it will be passed along: for example, one could type acs.fetch(geo=my.geos, endyear=2011, table.number="B08013", one.zero=T). See help(sum-methods) for more on this.

**Available data** By default, `acs.fetch()` will download the “Five-Year ACS” (`span=5`) data from the ACS (`dataset="acs"`), but these defaults can be changed by setting these options to other values.<sup>14</sup> As of version 2.0 of the package, `endyear` is a required option with no default; users must specify the *latest* year for the dataset they are seeking: for example, `endyear=2011` for the 2007–2011 ACS data (or, with `span=3`, for the 2009–2011 data...), or `endyear=2010` for the latest Decennial data (assuming that `dataset="sf1"` or `dataset="sf3"`, of course.)

At present, the Census API provides the following, all of which are available using `acs.fetch` with the proper combinations of `endyear`, `span`, and `dataset`:

- American Community Survey (`dataset="acs"`)
  - 5-Year Data: `endyear= 2009` through 2014 (i.e., six surveys, 2005–2009 through 2010–2014);
  - 3-Year Data: `endyear= 2012, 2013`;
  - 1-Year Data: `endyear= 2011, 2012, 2013, 2014`).
- Decennial Census Data
  - SF1/Short-Form (`dataset="sf1"`): `endyear= 1990, 2000, 2010`;
  - SF3/Long-Form (`dataset="sf3"`): `endyear= 1990, 2000`.<sup>15</sup>

See <http://www.census.gov/data/developers/data-sets.html> for more information about available data, including guidance about which geographies are provided for each dataset.

Downloading based on a table number is probably the most fool-proof way to get the data you want, but `acs.fetch()` will also accept a number of other arguments instead of `table.number`. Users can provide strings to search for in table names (e.g., `table.name="Age by Sex"` or `table.name="First Ancestry Reported"`) or keywords to find in the names of variables (e.g., `keyword="Male"` or `keyword="Haiti"`)—but be warned: given how many tables there are, you may get more matches than you expected and suffer from the “download overload” of fast-scrolling screens full of data.<sup>16</sup> On the other hand, if you know you want a specific variable or two (not a whole table, just a few columns of it—such as `variable="B05001_006"` or `variable=c("B16001_058", "B16001_059")`), you can ask for that with `acs.fetch(variable=variable.code, ...)`.

### 3.3.2 More descriptive variable names: `col.names=`

Variable names like `B01003_001` and `B05001_006` provide a great shorthand, and can be good for experienced users, but most of us prefer something more

---

<sup>14</sup>Users may set `span=1` or `span=3` for other ACS products, or `span=0` for Decennial data; similarly, use `dataset="sf1"` or `dataset="sf3"` for other census products.

<sup>15</sup>SF3 was discontinued after 2000 and replaced with the ACS.

<sup>16</sup>But don’t lose hope: see section 3.3.3 on the `acs.lookup()` tool, which can help with this problem.

descriptive. To help, the `acs.fetch()` function accepts a special argument called `col.names`, which can take any of the following values:

1. when `col.names="auto"` (the default), census variable codes are returned;
2. when `col.names` is given a character vector *the same length as the number of variables in the table*, these names will be used instead as variables for the new `acs` object; and
3. when `col.names="pretty"`, the function will use descriptive names for the variables (but beware: these can be quite long).

```
> ancestry=acs.fetch(geo=psrc, table.name="People Reporting Ancestry",
  endyear=2011, col.names="pretty")
```

```
> ancestry[, 20:30] # just a selection of rows -- it's a long table!
```

```
ACS DATA:
```

```
2007 -- 2011 ;
```

```
Estimates w/90% confidence intervals;
```

```
for different intervals, see confint()
```

```
King County      People Reporting Ancestry: Basque
```

```
1125 +/- 267
```

```
Kitsap County    41 +/- 34
```

```
Pierce County    210 +/- 113
```

```
Snohomish County 135 +/- 68
```

```
King County      People Reporting Ancestry: Belgian
```

```
2928 +/- 466
```

```
Kitsap County    428 +/- 196
```

```
Pierce County    781 +/- 197
```

```
Snohomish County 844 +/- 263
```

```
King County      People Reporting Ancestry: Brazilian
```

```
1716 +/- 519
```

```
Kitsap County    231 +/- 185
```

```
Pierce County    124 +/- 91
```

```
Snohomish County 221 +/- 97
```

```
King County      People Reporting Ancestry: British
```

```
17088 +/- 997
```

```
Kitsap County    1607 +/- 373
```

```
Pierce County    3943 +/- 573
```

```
Snohomish County 4735 +/- 599
```

```
King County      People Reporting Ancestry: Bulgarian
```

```
1659 +/- 409
```

```
Kitsap County    18 +/- 26
```

```
Pierce County    213 +/- 123
```

```
Snohomish County 444 +/- 248
```

```
King County      People Reporting Ancestry: Cajun
```

```
234 +/- 141
```

```
Kitsap County    49 +/- 41
```

Pierce County	222 +/- 117	
Snohomish County	140 +/- 92	
		People Reporting Ancestry: Canadian
King County	9996 +/- 984	
Kitsap County	1076 +/- 311	
Pierce County	3016 +/- 462	
Snohomish County	3694 +/- 527	
		People Reporting Ancestry: Carpatho Rusyn
King County	49 +/- 38	
Kitsap County	3 +/- 4	
Pierce County	0 +/- 92	
Snohomish County	0 +/- 92	
		People Reporting Ancestry: Celtic
King County	898 +/- 328	
Kitsap County	101 +/- 66	
Pierce County	263 +/- 101	
Snohomish County	207 +/- 121	
		People Reporting Ancestry: Croatian
King County	4577 +/- 647	
Kitsap County	596 +/- 243	
Pierce County	2334 +/- 496	
Snohomish County	743 +/- 234	
		People Reporting Ancestry: Cypriot
King County	0 +/- 92	
Kitsap County	0 +/- 92	
Pierce County	0 +/- 92	
Snohomish County	0 +/- 92	

### 3.3.3 the `acs.lookup()` function: finding the variables you want

Using `acs.fetch()` you can download all the data you need from the Census, provided you either *know* the variable codes or table numbers or are willing to make some educated guesses. This is a fine way to work, and it may be all you need to get started, but for more deliberate users, we've also developed a second lookup tool—known as `acs.lookup()`—to help identify the tables and variables they are interested in. As with the `geo.lookup()` tool, the results of `acs.lookup()` can be named, saved, modified, and eventually passed to `acs.fetch()` to get data.

**Finding the variables you want** `acs.lookup()` takes arguments similar to `acs.fetch`—in particular, `table.number`, `table.name`, and `keyword`, as well as `dataset` (optional, defaults to `dataset="acs"`), `endyear` (required, as with `acs.fetch`), and `span` (optional, defaults to `span=5`)—and searches for matches in the meta-data of the Census tables. When multiple search terms are passed to a given argument (e.g., `acs.lookup(endyear=2011, keyword=c("Female",`

"GED"))), the tool returns matches where *all* of the terms are found; similarly, when more than one lookup argument is used (e.g., `acs.lookup(endyear=2011, table.number="B01001", keyword="Female")`), the tool searches for matches that include all of these terms (i.e., terms are combined with a logical AND, not a logical OR). Like `acs.fetch`, string matches with `acs.lookup` are case sensitive by default, but users may change this by passing `case.sensitive=F` as an option.

```
> urdu=acs.lookup(keyword="Urdu", endyear=2011)
> urdu
An object of class "acs.lookup"
endyear= 2011 ; span= 5
```

```
results:
  variable.code table.number
1   B16001_057      B16001
2   B16001_058      B16001
3   B16001_059      B16001
                                                    table.name
1 Language Spoken at Home by Ability to Speak English for the Population 5+ Yrs
2 Language Spoken at Home by Ability to Speak English for the Population 5+ Yrs
3 Language Spoken at Home by Ability to Speak English for the Population 5+ Yrs
  variable.name
1
2 Urdu: Speak English 'very well'
3 Urdu: Speak English less than 'very well'
```

```
> age.by.sex=acs.lookup(table.name="Age by Sex", endyear=2011)
> age.by.sex
An object of class "acs.lookup"
endyear= 2011 ; span= 5
```

```
results:
  variable.code table.number
1   B01002_001      B01002
2   B01002_002      B01002
3   B01002_003      B01002
4   B23013_001      B23013 Median Age by Sex for Workers 16 to 64 Years
5   B23013_002      B23013 Median Age by Sex for Workers 16 to 64 Years
6   B23013_003      B23013 Median Age by Sex for Workers 16 to 64 Years
  variable.name
1 Median age -- Total:
2 Median age -- Male
3 Median age -- Female
4 Median age -- Total:
5 Median age-- Male
```

## 6 Median age-- Female

Arguments for the `acs.lookup` are documented in the help files (see `?acs.lookup`), but users unfamiliar with ACS variable nomenclature may want to spend a little time testing different search terms, keeping the following in mind:

- The `table.number` argument is fairly self-explanatory: it usually contains a six-character string, almost always starting with a “B” or “C”, followed by a five-digit number (e.g., “B01001” or “C02003”). For tables that include data from Puerto Rico, the table number may include the letters “PR” at the end (e.g., “B05001PR” for “Nativity and Citizenship Status in Puerto Rico”). *Note: For each `acs.lookup` search, only one table number is allowed.*
- Strings passed to the `table.name` argument provide search terms to match in the *table names* of the ACS: for example, “Sex” or “Age” or “Age by Sex”. Note: these include words that describe *types of categories*, not the categories themselves.
- The `keyword` argument contains terms to search for in the actual *variable names* of the table. Typically these include descriptive information on the *nominative categories* of the Census on Sex, Age, Race, Language, Ownership, and the like. Examples include “Male”, “Female”, “Black”, “Spanish”, “Subsaharan African”, “80 to 84 years”, “renter-occupied”, and so on. *Note: due to inconsistent capitalization rules, if you don't find the results, you expected, you may want to try again with `case.sensitive=F`.*
- Don't forget that `endyear` is a required argument for `acs.lookup`.
- While `dataset` and `span` are optional arguments, variable codes, table numbers, and table names may change from year to year or dataset to dataset, so it's best to specify them as well, just as you would do for `acs.fetch`.

To help keep it clear, as a rule of thumb: `table.name` tells you what *sort of categories* the table's variables contain, and `keyword` tells you what *particular categories* each specific variable includes. So if you want information on *all* races (or age groups or languages, etc.), use `table.name="Race"` (or `"Age"` or `"Language"`, etc.); if you only want a specific race (or age group or language, etc.), use `keyword="Asian"` (and so on).

**Manipulating and using `acs.lookup` objects** Since `acs.lookup` objects are valid objects in R, they can be named and saved (for example, `urdu` and `age.by.sex` above) and further manipulated by the user. Results contained within `acs.objects` can be subsetted (with `[square brackets]`), and even combined (with either `c()` or `+`—both function the same way) to create new `acs.lookup` objects.

```

> workers.age.by.sex=age.by.sex[4:6]
> my.vars=workers.age.by.sex+urdu
> # could also be:
> # my.vars=c(workers.age.by.sex, urdu)
> my.vars
An object of class "acs.lookup"
endyear= 2011 ; span= 5

```

results:

	variable.code	table.number
4	B23013_001	B23013
5	B23013_002	B23013
6	B23013_003	B23013
41	B16001_057	B16001
51	B16001_058	B16001
61	B16001_059	B16001

	table.name
4	Median Age by Sex for Workers 16 to 64 Years
5	Median Age by Sex for Workers 16 to 64 Years
6	Median Age by Sex for Workers 16 to 64 Years
41	Language Spoken at Home by Ability to Speak English for the Population 5+ Yrs
51	Language Spoken at Home by Ability to Speak English for the Population 5+ Yrs
61	Language Spoken at Home by Ability to Speak English for the Population 5+ Yrs

  

	variable.name
4	Median age -- Total:
5	Median age-- Male
6	Median age-- Female
41	Urdu:
51	Urdu: Speak English 'very well'
61	Urdu: Speak English less than 'very well'

```

> acs.fetch(geography=psrc, endyear=2011, variable=my.vars)

```

ACS DATA:

2007 -- 2011 ;

Estimates w/90% confidence intervals;  
for different intervals, see confint()

	B23013_001	B23013_002	B23013_003
King County, Washington	39.6 +/- 0.2	39.5 +/- 0.2	39.6 +/- 0.2
Kitsap County, Washington	41.2 +/- 0.3	40.3 +/- 0.3	42.3 +/- 0.4
Pierce County, Washington	39.8 +/- 0.2	39.6 +/- 0.2	40.1 +/- 0.2
Snohomish County, Washington	41.1 +/- 0.2	40.9 +/- 0.2	41.3 +/- 0.3
	B16001_057	B16001_058	B16001_059
King County, Washington	1735 +/- 557	1308 +/- 454	427 +/- 170
Kitsap County, Washington	75 +/- 99	75 +/- 99	0 +/- 92
Pierce County, Washington	219 +/- 189	204 +/- 178	15 +/- 27
Snohomish County, Washington	1179 +/- 520	858 +/- 329	321 +/- 267

>

Note that these “acs.lookup” objects can also be passed as variables to `acs.fetch` with different (new) values for `endyear` and `span`:

```
> acs.fetch(geography=psrc, endyear=2014, variable=my.vars)
```

ACS DATA:

2010 -- 2014 ;

Estimates w/90% confidence intervals;

for different intervals, see `confint()`

	B23013_001	B23013_002	B23013_003
King County, Washington	39.7 +/- 0.1	39.8 +/- 0.2	39.6 +/- 0.2
Kitsap County, Washington	41 +/- 0.3	39.6 +/- 0.4	42.6 +/- 0.5
Pierce County, Washington	40 +/- 0.2	39.7 +/- 0.2	40.4 +/- 0.3
Snohomish County, Washington	41.3 +/- 0.2	41.4 +/- 0.2	41.1 +/- 0.2
	B16001_057	B16001_058	B16001_059
King County, Washington	2343 +/- 527	1880 +/- 425	463 +/- 168
Kitsap County, Washington	62 +/- 91	62 +/- 91	0 +/- 28
Pierce County, Washington	108 +/- 161	77 +/- 106	31 +/- 56
Snohomish County, Washington	1123 +/- 488	860 +/- 318	263 +/- 211

```
> acs.fetch(geography=psrc, endyear=2014, span=1, variable=my.vars)
```

ACS DATA:

2014 ;

Estimates w/90% confidence intervals;

for different intervals, see `confint()`

	B23013_001	B23013_002	B23013_003
King County, Washington	39.4 +/- 0.2	39.6 +/- 0.2	39.1 +/- 0.4
Kitsap County, Washington	40 +/- 0.7	38.1 +/- 1.4	41.8 +/- 1.2
Pierce County, Washington	39.3 +/- 0.4	39.2 +/- 0.5	39.4 +/- 0.7
Snohomish County, Washington	41 +/- 0.5	41.2 +/- 0.5	40.7 +/- 0.7
	B16001_057	B16001_058	B16001_059
King County, Washington	2844 +/- 1478	2413 +/- 1158	431 +/- 403
Kitsap County, Washington	NA +/- NA	NA +/- NA	NA +/- NA
Pierce County, Washington	NA +/- NA	NA +/- NA	NA +/- NA
Snohomish County, Washington	1856 +/- 1247	1581 +/- 1076	275 +/- 258

>

And, in this way, once the Census has released data for 2015 users may begin to download it *even before the acs package has been updated*:

```
> acs.fetch(geography=psrc, endyear=2015, variable=my.vars)
```

Error in file(file, "rt") : cannot open the connection

```
> # error now, but when the data is available through the API
```

```
> # this will actually work!!
```



## 4 Exporting Data

In the future, versions of the `acs` package will include improved export functions to allow users to save `acs` data in a variety of formats. For now, however, users wishing to export data for use in spreadsheets or other program can make use of the existing export functions, such as `write.csv`, along with the package's `estimate`, `standard.error`, and `confint` functions. Thus, to save the estimates, standard errors, and a 90% confidence interval as three different `.csv` spreadsheets:

```
> write.csv(estimate(ancestry), file="./ancestry_estimate.csv")
> write.csv(standard.error(ancestry), file="./ancestry_error.csv")
> write.csv(confint(ancestry, level=.90), file="./ancestry_confint.csv")
```

Depending on the shape you ideally want the data to take, you may want to first create a dataframe from these various elements—a first column of estimate, a second column of 90% MOEs, for example—and then save that:

```
> urdu.speakers=acs.fetch(geography=c(psrc, north.mercer.island.plus),
  variable=urdu[1], endyear=2011, col.names="Speak Urdu")
> urdu.speakers
ACS DATA:
2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
                                Speak Urdu
King County, Washington         1735 +/- 557
Kitsap County, Washington        75 +/- 99
Pierce County, Washington        219 +/- 189
Snohomish County, Washington    1179 +/- 520
North Mercer Island Tracts       0 +/- 159.348674296337

> my.data=data.frame(estimate(urdu.speakers),
  1.645*standard.error(urdu.speakers))
> colnames(my.data)=c("Estimate","90% MOE")
> my.data
                                Estimate  90% MOE
King County, Washington         1735 557.0000
Kitsap County, Washington        75 99.0000
Pierce County, Washington        219 189.0000
Snohomish County, Washington    1179 520.0000
North Mercer Island Tracts       0 159.3487

> write.csv(my.data, file="./urdu.csv")
>
```

## 5 Additional Resources

The `acs.R` package is hosted on the CRAN repository, where updates will appear from time to time. For additional guidance and examples, users are advised to review the complete documentation at (<http://cran.r-project.org/web/packages/acs/index.html>), which can also be accessed in an R session via the `help` function.

Additional insights on the general object-oriented approach of the package may be found in my 2011 article, “acs.R: an R Package for Neighborhood-Level Data from the U.S. Census” ([http://papers.ssrn.com/sol3/papers.cfm?abstract\\_id=2171390](http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2171390)). In addition, the “CityState” website <http://eglenn.scripts.mit.edu/citystate/> will continue to include updates, patches, worked examples, and more. And finally, users may subscribe to a mailing list at <http://mailman.mit.edu/mailman/listinfo/acs-r> to keep in touch about the ongoing development of the package, including information on ongoing development; user questions, technical assistance, and new feature requests; and additional updates.

## A A worked example using blockgroup-level data and nested combined geo.sets

To showcase how the package can create new census geographies based on blockgroups—the smallest census geographies provided via the Census API—we can use the following example from Middlesex County in Massachusetts.

### A.1 Making the geo.set

To gather data on all the block groups for tract 387201, we create a new geo like this:

```
> my.tract=geo.make(state="MA", county="Middlesex",
  tract=387201, block.group="*", check=T)
Testing geography item 1: Tract 387201, Blockgroup *,
  Middlesex County, Massachusetts .... OK.
>
```

This might be a useful first step, especially if I didn't know how many block groups there were in the tract, or what they were called. Also, note that `check=T` is not required, but can often help ensure you are dealing with valid geos.

If we then wanted to get very basic info on these block groups—say, table number “B01003” (Total Population), we use:

```
> total.pop=acs.fetch(geo=my.tract, endyear=2011, table.number="B01003")
> total.pop
ACS DATA:
2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
      B01003_001
Block Group 1 2681 +/- 319
Block Group 2 952 +/- 213
Block Group 3 1010 +/- 156
Block Group 4 938 +/- 214
>
```

Here we can see that the `block.group="*"` has yielded the actual four block groups for the tract.<sup>17</sup> Now, if instead of wanting all of them, we only wanted the first two, we could just type:

```
> my.bgs=geo.make(state="MA", county="Middlesex",
  tract=387201, block.group=1:2, check=T)
```

---

<sup>17</sup>A similar approach can help find the names of all tracts in a county, for example: `acs.fetch(geography=geo.make(state="MA", county="Middlesex", tract="*"), table.number="B01001")` returns a list of all 300+ tracts in the county, with estimates of total population.

```

Testing geography item 1: Tract 387201, Blockgroup 1,
  Middlesex County, Massachusetts .... OK.
Testing geography item 2: Tract 387201, Blockgroup 2,
  Middlesex County, Massachusetts .... OK.
>

```

And then:

```

> bg.total.pop=acs.fetch(geo=my.bgs, endyear=2011, table.number="B01003")
> bg.total.pop
ACS DATA:
  2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
      B01003_001
Block Group 1 2681 +/- 319
Block Group 2 952 +/- 213
>

```

Now, if we wanted to add in some blockgroups from tract 387100 (a.k.a. "tract 3871"—but remember: we need those trailing zeroes)—say, blockgroups 2 and 3—we could enter:

```

> my.bgs=my.bgs+geo.make(state="MA", county="Middlesex",
  tract=387100, block.group=2:3, check=T)
Testing geography item 1: Tract 387100, Blockgroup 2,
  Middlesex County, Massachusetts .... OK.
Testing geography item 2: Tract 387100, Blockgroup 3,
  Middlesex County, Massachusetts .... OK.

```

And then:

```

> acs.fetch(geo=my.bgs, endyear=2011, table.number="B01003")
ACS DATA:
  2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
      B01003_001
Block Group 1, Census Tract 3872.01, Middlesex County, ... 2681 +/- 319
Block Group 2, Census Tract 3872.01, Middlesex County, ... 952 +/- 213
Block Group 2, Census Tract 3871, Middlesex County, ... 827 +/- 171
Block Group 3, Census Tract 3871, Middlesex County, ... 1821 +/- 236
>

```

## A.2 Using combine=T to make a neighborhood

Next, to showcase the real power of `geo.sets`: let's say we don't just want to get data on the four blockgroups, but I want to *combine* them into a

single new geographic entity—say, a neighborhood called “Turkey Hill.” Before downloading, we could simply say:

```
> combine(my.bgs)=T
> combine.term(my.bgs)="Turkey Hill"
> acs.fetch(geo=my.bgs, endyear=2011, table.number="B01003")
ACS DATA:
2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
      B01003_001
Turkey Hill 6281 +/- 481.733328720362
>
```

And *voilà!*, the package sums the estimates and deals with the margins of error, so we don’t need to get our hands dirty with square roots and standard errors and all that messy stuff.

### A.3 Even more complex geo.sets

We can even create interesting nested geo.sets, where some of the lower levels are combined, and others are kept distinct:

```
> more.bgs=c(my.bgs, geo.make(state="MA",
  county="Middlesex", tract=370300, block.group=1:2, check=T),
  geo.make(state="MA", county="Middlesex", tract=370400,
  block.group=1:3, combine=T, combine.term="Quirky Hill", check=T))
Testing geography item 1: Tract 370300, Blockgroup 1, .... OK.
Testing geography item 2: Tract 370300, Blockgroup 2, .... OK.
Testing geography item 1: Tract 370400, Blockgroup 1, .... OK.
Testing geography item 2: Tract 370400, Blockgroup 2, .... OK.
Testing geography item 3: Tract 370400, Blockgroup 3, .... OK.
> acs.fetch(geo=more.bgs, endyear=2011, table.number="B01003",
  col.names="pretty")
ACS DATA:
2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
Total Population: Total
Turkey Hill          6281 +/- 481.733328720362
Block Group 1, Census Tract 3703  315 +/- 132
Block Group 2, Census Tract 3703  1460 +/- 358
Quirky Hill          2594 +/- 487.719181496894
>
```

We can even create a `geo.set` that bundles different levels of census geography— for example, our two neighborhoods (“Turkey Hill” and “Quirky Hill”), plus some data for comparison on the entire county and state level.

```

> neighborhood.geos=c(more.bgs[c(1,3)],
  geo.make(state="MA", county="Middlesex"),
  geo.make(state="MA"))
> acs.fetch(geography=neighborhood.geos, endyear=2011,
  table.number="B01003", col.names="pretty")
ACS DATA:
2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()

```

	Total Population: Total
Turkey Hill	6281 +/- 481.733328720362
Quirky Hill	2594 +/- 487.719181496894
Middlesex County, Massachusetts	1491762 +/- 0
Massachusetts	6512227 +/- 0

```

>

```

Note that this geo.set can now be used again and again to download and analyze many different variables for these same geographies.

#### A.4 Gathering neighborhood data on transit mode-share

As a final example, let's look for some data on commuting choices for these two neighborhoods, compared to the county and state. If we don't know what census variables we want, we can use the `acs.lookup` function to search for likely candidates. Let's see which variables use the word "Bicycle":

```

> acs.lookup(keyword="Bicycle", endyear=2011)
An object of class "acs.lookup"
endyear= 2011 ; span= 5

results:
  variable.code table.number
1   B08006_014   B08006
2   B08006_031   B08006
3   B08006_048   B08006
4   B08301_018   B08301
5   B08406_014   B08406
6   B08406_031   B08406
7   B08406_048   B08406
8   B08601_018   B08601

```

	table.name
1	Sex of Workers by Means of Transportation to Work
2	Sex of Workers by Means of Transportation to Work
3	Sex of Workers by Means of Transportation to Work
4	Means of Transportation to Work
5	Sex of Workers by Means of Transportation to Work for Workplace Geography

```

6 Sex of Workers by Means of Transportation to Work for Workplace Geography
7 Sex of Workers by Means of Transportation to Work for Workplace Geography
8           Means of Transportation to Work for Workplace Geography
      variable.name
1           Bicycle
2   Male: Bicycle
3   Female: Bicycle
4           Bicycle
5           Bicycle
6   Male: Bicycle
7   Female: Bicycle
8           Bicycle
>

```

We've quickly narrowed a few thousand variables down to just 8. As is common with the ACS, there are a number of tables that relate to the topic we are interested in (means of transportation), often cross-tabulated with other topics. The simplest one seems to be the fourth in the list, "Means of Transportation to Work," from table number B08301. Let's look at all the variables there, just to be sure:

```

> acs.lookup(table.number="B08301", endyear=2011)
An object of class "acs.lookup"
endyear= 2011 ; span= 5

results:
  variable.code table.number           table.name
1   B08301_001      B08301 Means of Transportation to Work
2   B08301_002      B08301 Means of Transportation to Work
3   B08301_003      B08301 Means of Transportation to Work
4   B08301_004      B08301 Means of Transportation to Work
5   B08301_005      B08301 Means of Transportation to Work
... [abbreviated for space]

                                variable.name
1                                Total:
2                                Car, truck, or van:
3                                Car, truck, or van: Drove alone
4                                Car, truck, or van: Carpooled:
5   Car, truck, or van: Carpooled: In 2-person carpool
... [abbreviated for space]
>

```

This seems to be what we want, including data on people who drove to work alone, biked, took public transit, and so on for 20 different modes (as well as the all important "Total" on the first line, which we will need for percentages).

For our purposes, let's look at just a few of these variables: drove alone, public transportation, biking, and the total population from the table.<sup>18</sup> We can subset these and save them as a new `acs.lookup` object, and pass them right on to fetch some data.

```
> transit.vars=acs.lookup(table.number="B08301")[c(1,3,10,18),
  endyear=2011]
> transit.vars
An object of class "acs.lookup"
endyear= 2011 ; span= 5
```

results:

	variable.code	table.number	table.name
1	B08301_001	B08301	Means of Transportation to Work
3	B08301_003	B08301	Means of Transportation to Work
10	B08301_010	B08301	Means of Transportation to Work
18	B08301_018	B08301	Means of Transportation to Work

  

	variable.name
1	Total:
3	Car, truck, or van: Drove alone
10	Public transportation (excluding taxicab):
18	Bicycle

```
> transit.data=acs.fetch(geography=neighborhood.geos,
  variable=transit.vars, endyear=2011,
  col.names=c("Total", "Drove Alone", "Public Transit", "Biked"))
> transit.data
```

ACS DATA:

```
2007 -- 2011 ;
Estimates w/90% confidence intervals;
for different intervals, see confint()
```

	Total
Turkey Hill	3159 +/- 405.076535978079
Quirky Hill	1891 +/- 380.596899619532
Middlesex County, Massachusetts	773894 +/- 3339
Massachusetts	3202521 +/- 8062
	Drove Alone
Turkey Hill	2687 +/- 352.326553072572
Quirky Hill	1068 +/- 301.584150777192
Middlesex County, Massachusetts	539042 +/- 3602
Massachusetts	2316985 +/- 8271
	Public Transit
Turkey Hill	110 +/- 133.285408053545

<sup>18</sup>Note the importance of the last of these variables: when computing percentages for ACS data, always use the totals from the particular table, not from some other "Total population" table.



```

Quirky Hill                333 +/- 133.007518584477
Middlesex County, Massachusetts 82883 +/- 1931
Massachusetts              291160 +/- 3799
                           Biked
Turkey Hill                0 +/- 190
Quirky Hill                40 +/- 103.009708280336
Middlesex County, Massachusetts 9661 +/- 725
Massachusetts              21938 +/- 1195
>

```

Since these are raw counts, and we might be more interested in percentages, we can use the special math functions of the `acs` package to divide the last three columns by the first. (The division function will automatically deal with both estimates and standard errors.) In some cases, division on `acs` objects is quite simple: something like `transit.data[,2]/transit.data[,1]` would convert the second column from counts to percentages. We can try that here, as follows:

```

> transit.data[,2]/transit.data[,1]
ACS DATA:
2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
                                ( Drove Alone : Total )
Turkey Hill                    0.850585628363406 +/- 0.155998230670757
Quirky Hill                    0.564780539397144 +/- 0.195848029196112
Middlesex County, Massachusetts 0.696532083205193 +/- 0.00554027354078655
Massachusetts                  0.723487839736258 +/- 0.00316025915803343
Warning message:
In transit.data[, 2]/transit.data[, 1] :
  ** using the more conservative formula for ratio-type
  dividing, which does not assume that numerator is a subset of
  denominator; for more precise results when seeking a proportion
  and not a ratio, use divide.acs(..., method="proportion") **
>

```

In this case, however, as the warning notes, this is actually slightly wrong: since this should in fact be a “proportion-type” division (and not a “ratio-type” division—see `?divide.acs`), we don’t want standard division with `/`, but instead must use the package’s special `acs.divide` function. This can be called on each column of our data with R’s standard `apply` function, which has been adapted to work on `acs` objects.

```

> apply(transit.data[,2:4], MARGIN=1, FUN=divide.acs,
        denominator=transit.data[,1], method="proportion",
        verbose=F)
ACS DATA:

```

```

2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
                                ( Drove Alone / Total )
Turkey Hill                    0.850585628363406 +/- 0.0233001603324679
Quirky Hill                    0.564780539397144 +/- 0.111865144361133
Middlesex County, Massachusetts 0.696532083205193 +/- 0.00355414596064069
Massachusetts                  0.723487839736258 +/- 0.00183110720072149
                                ( Public Transit / Total )
Turkey Hill                    0.034821145932257 +/- 0.0419553490713477
Quirky Hill                    0.176097303014278 +/- 0.0607546663302448
Middlesex County, Massachusetts 0.107098646584674 +/- 0.00245201395451528
Massachusetts                  0.0909158753369611 +/- 0.00116396486009957
                                ( Biked / Total )
Turkey Hill                    0 +/- 0.0601456157011713
Quirky Hill                    0.0211528291909043 +/- 0.0546397830403873
Middlesex County, Massachusetts 0.0124836217879968 +/- 0.000938367861201701
Massachusetts                  0.00685022830451385 +/- 0.000373541799644013
>

```

Note in passing that the resulting *estimates* are the same as in the previous division, but that there errors are slightly different as a result of the proportion-type operation.<sup>19</sup>

Now we can see something interesting in our data: not only do far more people in Turkey Hill drive alone (and far fewer take public transit) than in Quirky Hill (or even in the county or state), the differences seem far beyond the report margin of errors.

---

<sup>19</sup>If you don't set `verbose=F`, the function also returns some warnings—the first two just to let you know that proportion-division is not the same as ratio-division; the third lets you know that in one case, the function defaulted to ratio-style division as per Census guidance.