Working with acs.R
Introductory Notes

Ezra Haber Glenn, AICP
Public Planning, Research, & Implementation, Inc.

January, 2013

1 Purpose

These notes are intended to accompany my initial version of the updated `acs.R` package, developed for the Puget Sound Regional Council. In the near future, these draft notes will be replaced with a more detailed User Guide with additional sections on using `R` and data analysis with the `acs.R` package, but for now as we test out these scripts and functions, I wanted to be able to let you play with them as well, to be sure they meet your needs.

2 Key Improvements

Consistent with the outlined scope of services, I've added new functions to the `acs.R` package to allow users to create their own geographies by combining existing ones provided by the Census. In addition to automating the process of downloading, importing, and storing data from the American Community Survey for these new geographies (including proper statistical techniques for dealing with estimates and standard errors), the package also 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.

3 A Quick Guide to `R`

To come.

4 Installing the Package

Eventually, the revised package will be hosted on the CRAN repository, replacing the current version of `acs.R`, and the User Guide will include a section on installing `R` and adding the package. For now, I am providing the package as a “zipped tarball”—still a complete package, but not quite ready for uploading to the site. There may be some bugs still to find, and the documentation needs work, but it can be installed just like any other package. Simply start R and then type:

```
> # do this once, you never need to do it again
> install.packages(pkgs = "acs_1.0.tar.gz")
inferring 'repos = NULL' from the file name
* 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

>
```

4.1 Additional Required Packages, Installation Trouble

The acs.R package depends on a few other fairly common `R` packages: `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.

5 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://www.census.gov/developers/tos/key_request.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,[1] 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")
>
```

6 Working with the New Functions

6.1 Overview

I'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 4 and 5);

[1]or at least those that require interaction with the API, such as `acs.fetch()` and the `check=` option for `geo.make()`—the `xxx.lookup` functions work fine without a key, and can even be used when you are off-line.

2. create a `geo.set` using the `geo.make()` function (see section 6.2);
3. optionally, use the `acs.lookup()` function to explore the variables you may want to download (see section 6.3.3 on page 17) ;
4. use the `acs.fetch()` function to download data for your new geography (see section 6.3.1 on page 14); and then
5. use the existing functions in the package to work with your data (additional sections to come).

As a teaser, here you can see one single command that would 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="*"), 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. . .).

6.2 User-Specific Geographies

6.2.1 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.[2] 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"
```

[2]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 will 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."

```
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.[3] 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 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;[4] 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]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.

[4]But don't fret: see section 6.2.7 on page 10.

6.2.2 But where's the data. . . ?

Note that these new `geo.set`s are simply placeholders for geographic entities—they do not actually contain any census data *about* these places. Be patient (or jump ahead to section 6.3 on page 13).

6.2.3 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 6.2.1 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.set`s 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"`).[5] 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()` 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]]
```

[5]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.

```
"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 6.2.5 on page 9), regardless of whether the constituent parts are nested sets.[6]

```
> 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
> 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"
```

[6]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.

```
  .. ..$ :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.)

6.2.4 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=`[7].

```
> 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
```

[7]or `combine()<-` and `combine.term()<-`, for `R` traditionalists. . .

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

6.2.5 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 helper function named `flatten.geo.set()`, 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.

6.2.6 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.

6.2.7 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").[8]

```
> 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] 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.

```
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.[9]

```
> 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)
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")
```

[9] 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: 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!
```

6.3 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.[10]

[10] 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")`.

6.3.1 `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 5 on page 3)[11], the call is quite simple:

```
> # table B01003: "Total Population"
> acs.fetch(geography=psrc, 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, 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
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",
                col.names=c("Total","Male","Female"))
ACS DATA:
```

[11] And if you haven't, you can simply add a `key=` argument each time.

```
 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
```

By default, `acs.fetch()` will download the ACS data from 2007–2011 (the "Five-Year ACS"), but the functions includes options for `endyear=` and `span=`. At present, the API only provides the five-year data for 2006–2010 and 2007–2011, but as more data is added the function will be able to download other years and products.[12]

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.[13] 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")`),

[12]To address this lack of data, as well as PSRC's desire to provide summary level 155 data (which is not available through the API), I am working on a new helper function to simplify the download process through FactFinder, so that a user specifying a `geo.set` and a table number or `acs.lookup` object would be generate a URL "deep-linking" to downloadable results from FactFinder.

[13]But don't lose hope: see section 6.3.3 on the `acs.lookup()` tool, which can help with this problem.

you can ask for that with `acs.fetch(variable=`*variable.code*`, ...)`.

6.3.2 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 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",
     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()
                 People Reporting Ancestry:  Basque
King County      1125 +/- 267
Kitsap County    41 +/- 34
Pierce County    210 +/- 113
Snohomish County 135 +/- 68
                 People Reporting Ancestry:  Belgian
King County      2928 +/- 466
Kitsap County    428 +/- 196
Pierce County    781 +/- 197
Snohomish County 844 +/- 263
                 People Reporting Ancestry:  Brazilian
King County      1716 +/- 519
Kitsap County    231 +/- 185
Pierce County    124 +/- 91
Snohomish County 221 +/- 97
                 People Reporting Ancestry:  British
King County      17088 +/- 997
Kitsap County    1607 +/- 373
Pierce County    3943 +/- 573
Snohomish County 4735 +/- 599
                 People Reporting Ancestry:  Bulgarian
King County      1659 +/- 409
```

```
Kitsap County    18 +/- 26
Pierce County    213 +/- 123
Snohomish County 444 +/- 248
                 People Reporting Ancestry:  Cajun
King County      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
```

6.3.3 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.

`acs.lookup()` takes arguments similar to `acs.fetch`—in particular, `table.number`, `table.name`, and `keyword`, as well as `endyear` and `span`—and searches for matches in the meta-data of the Census tables. The results can then be inspected, subsetted (with `[square brackets]`), and combined (with `c()` or `+`) to create `acs.lookup` objects for you to use.

```
> urdu=acs.lookup(keyword="Urdu")
> 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                                       Urdu:
2           Urdu: Speak English 'very well'
3  Urdu: Speak English less than 'very well'

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

results:
  variable.code table.number                                      table.name
1    B01002_001       B01002                               Median Age by Sex
2    B01002_002       B01002                               Median Age by Sex
3    B01002_003       B01002                               Median Age by Sex
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
```

```
> workers.age.by.sex=age.by.sex[4:6]
> acs.fetch(geo=psrc, variable=workers.age.by.sex)
ACS DATA:
 2007 -- 2011 ;
  Estimates w/90% confidence intervals;
  for different intervals, see confint()
                 B23013_001   B23013_002   B23013_003
King County      39.6 +/- 0.2 39.5 +/- 0.2 39.6 +/- 0.2
Kitsap County    41.2 +/- 0.3 40.3 +/- 0.3 42.3 +/- 0.4
Pierce County    39.8 +/- 0.2 39.6 +/- 0.2 40.1 +/- 0.2
Snohomish County 41.1 +/- 0.2 40.9 +/- 0.2 41.3 +/- 0.3
```

(The finished User Guide will have additional guidance on searches with this function, but for now I just wanted to call your attention to it and show how interacts with the `acs.fetch()` tool.)

7 Analyzing Data with the `acs.R` Package

To come. Also, for now, users may want to review the existing documentation for the `acs.R` package (`http://cran.r-project.org/web/packages/acs/index.html`) as well as my 2011 article, "acs.R: an R Package for Neighborhood-Level Data from the U.S. Census" (`http://dusp.mit.edu/sites/all/files/attachments/publication/glenn_acs_cupum_jou.pdf`).