Preliminaries

This page is an IPython notebook. If you have the actual notebook file (not a PDF or HTML file generated from it), *and* you have a version of IPython installed that uses the same Python as the LSST stack, you can run all of the Python commands on this page. You'll need to change the DATA_DIR variable in the first line to something appropriate for you, and run hscProcessCcd.py on at least one CCD first (and if it's not the same CCD I used here, you'll have to change that variable as well).

The Butler

The Butler is the object that manages all of the pipeline's inputs and outputs. It manages the directory structure, so you don't have to - while you can read the files directly, it's always better to use the Butler if you can. To use it, you'll need that basic directory structure to be setup already, including a couple of special files. In this tutorial, we'll assume that's already been done.

In [1]:
```
import lsst.daf.persistence
DATA_DIR = "/home/jbosch/HSC/tutorial/data"
butler = lsst.daf.persistence.Butler(DATA_DIR)
```

Using the butler to get a dataset is just a matter of knowing the name of the dataset its "data ID", a dictionary that describes the particular unit of data you want. The dataset name for a CCD-level calibrated image is "calexp", and for a data ID, we provide the visit and CCD numbers:

In [2]:
```
exposure = butler.get("calexp", visit=1238, ccd=40, immediate=True)
```

We could also pass the data ID as a dict, which is useful if we want to reuse it:

In [3]:
```
dataId = {"visit": 1238, "ccd": 40}
exposure = butler.get("calexp", dataId, immediate=True)
```

The `immediate=True` argument isn't strictly necessary, but it can help make bugs in later code harder to track down. Without it, instead of returning the thing you want, the Butler returns a proxy object that mimics the object you want, and doesn't actually load that dataset until you try to use it. Most of the time, that just causes confusion, so it's best to just use `immediate=True` to disable the proxy behavior and have the Butler load and return the dataset immediately.

The Python `exposure` object we get from the butler in this case is an instance of `lsst.afw.image.ExposureF` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1image_1_1_exposure.html), which is a combination of several things, including: - `getMaskedImage()` returns an `lsst.afw.image.MaskedImageF` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1image_1_1_masked_image.html), which is itself a collection of three images: - `getImage()` returns an `lsst.afw.image.ImageF` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1image_1_1_image.html), containing the science image - `getMask()` returns an `lsst.afw.image.MaskU` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1image_1_1_mask.html), containing bad pixel masks - `getVariance()` returns an `lsst.afw.image.ImageF` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1image_1_1_image.html), containing the per-pixel variance - `getPsf()` returns an `lsst.afw.detection.Psf` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1detection_1_1_psf.html), our model for the point-spread function - `getWcs()` returns an `lsst.afw.image.Wcs` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1image_1_1_wcs.html), with the image's astrometric registration - `getCalib()` returns an `lsst.afw.image.Calib`

(http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1afw_1_1image_1_1_calib.html), with the image's photometric calibration - `getMetadata()` returns an `lsst.daf.base.PropertySet` (http://hsca.ipmu.jp/doxygen/latest/classlsst_1_1daf_1_1base_1_1_property_set.html), with miscellaneous other metadata

An `Exposure` is saved as a multi-extension FITS file. The first three HDUs are just the first three image planes

A `visit+ccd` data ID is also used by the main catalog of single frame measurements, which is the `src` dataset:

```
In [4]: catalog = butler.get("src", dataId, immediate=True)
```

The `catalog` object here is an instance of `lsst.afw.table.SourceCatalog`. We'll return to it later in much greater detail.

Important Data Products

This isn't an exhaustive list of all data products, but it has the ones you'll use most of the time. You can look in obs_subaru/policy/HsfMapper.paf for that, but beware that there's a lot of old things there too that are no longer used. While we won't cover most of these in the tutorial, the Name and Data ID Keys columns give you everything you need to know to get these from the Butler, and you'll note that most of them are either `lsst.afw.image.ExposureF` and `lsst.afw.table.SourceCatalog`, and that means most of the tutorials below will apply to them indirectly.

Name	**Data ID Keys**	**Type**	**Produced By**	**Description**
calexp	visit, ccd	lsst.afw.image.ExposureF	processCcd.py/reduceFrames.py	background-subtracted exposure, with Wcs, Psf, Calib, etc.
calexpBackground	visit, ccd	lsst.afw.math.BackgroundList	processCcd.py/reduceFrames.py	background subtracted from calexp
src	visit, ccd	lsst.afw.table.SourceCatalog	processCcd.py/reduceFrames.py	main single-frame catalog
icSrc	visit, ccd	lsst.afw.table.SourceCatalog	processCcd.py/reduceFrames.py	bright sources used for initial single-frame calibration
forced_sr	visit, ccd, tract	lsst.afw.table.SourceCatalog	forcedPhotCcd.py	forced photometry, using coadd positions and shapes
calibrated_src	visit, ccd, tract	lsst.afw.table.SourceCatalog	calibrateCatalog.py	über-calibrated src
calibrated_exp	visit, ccd, tract	lsst.afw.image.ExposureF	calibrateExposure.py	über-calibrated calexp

deepCoadd_calexp	tract, patch, filter	lsst.afw.image.ExposureF	processCoadd.py/stack.py	background-subtracted coadd, with Wcs, Psf, Calib, ApCorr, etc.
deepCoadd_calexpBackground	tract, patch, filter	lsst.afw.math.BackgroundList	processCoadd.py/stack.py	background subtracted from deepCoadd_calexp
deepCoadd_src	tract, patch, filter	lsst.afw.table.SourceCatalog	processCoadd.py/stack.py	main coadd catalog, done separately in each band
deepCoadd_forced_src	tract, patch, filter	lsst.afw.table.SourceCatalog	forcedPhotCoadd.py	forced photometry, using one band as reference for centroids, shapes

Catalogs

Schemas, Fields, and Columns

The most important thing you can do to understand what's in a catalog is to look at its schema:

In [5]:
```
print catalog.schema
```

```
Schema(
    (Field['L'](name="id", doc="unique ID"), Key<L>(offset=0, nElements=1)),
    (Field['Coord'](name="coord", doc="position in ra/dec", units="IRCS; radians"), Key<Coord>(offset=8, nElements=2)),
    (Field['L'](name="parent", doc="unique ID of parent source"), Key<L>(offset=24, nElements=1)),
    (Field['Flag'](name="calib.detected", doc="Source was detected as an icSrc"), Key['Flag'](offset=32, bit=0)),
    (Field['Flag'](name="calib.psf.candidate", doc="Flag set if the source was a candidate for PSF determination, as determined by the"), Key['Flag'](offset=32, bit=1)),
    (Field['Flag'](name="calib.psf.used", doc="Flag set if the source was actually used for PSF determination, as determined by the"), Key['Flag'](offset=32, bit=2)),
    (Field['Flag'](name="calib.psf.reserved", doc="Flag set if the source was selected as a PSF candidate, but was reserved from the PSF fitting."), Key['Flag'](offset=32, bit=3)),
    (Field['Flag'](name="flags.negative", doc="set if source was detected as significantly negative"), Key['Flag'](offset=32, bit=4)),
    (Field['I'](name="deblend.nchild", doc="Number of children this object has (defaults to 0)"), Key<I>(offset=40, nElements=1)),
    (Field['Flag'](name="deblend.deblended-as-psf", doc="Deblender thought this source looked like a PSF"), Key['Flag'](offset=32, bit=5)),
```

```
    (Field['PointD'](name="deblend.psf-center", doc="If deblended-as-psf, the PSF cent
roid"), Key<PointD>(offset=48, nElements=2)),
    (Field['D'](name="deblend.psf-flux", doc="If deblended-as-psf, the PSF flux"), Key
<D>(offset=64, nElements=1)),
    (Field['Flag'](name="deblend.too-many-peaks", doc="Source had too many peaks; only
the brightest were included"), Key['Flag'](offset=32, bit=6)),
    (Field['Flag'](name="deblend.parent-too-big", doc="Parent footprint covered too ma
ny pixels"), Key['Flag'](offset=32, bit=7)),
    (Field['Flag'](name="deblend.failed", doc="Deblending failed on source"), Key['Fla
g'](offset=32, bit=8)),
    (Field['Flag'](name="deblend.skipped", doc="Deblender skipped this source"), Key
['Flag'](offset=32, bit=9)),
    (Field['Flag'](name="deblend.ramped.template", doc="This source was near an image
edge and the deblender used "ramp" edge-handling."), Key['Flag'](offset=32, bit=10)),
    (Field['Flag'](name="deblend.patched.template", doc="This source was near an image
edge and the deblender used "patched" edge-handling."), Key['Flag'](offset=32, bit=1
1)),
    (Field['Flag'](name="deblend.has.stray.flux", doc="This source was assigned some s
tray flux"), Key['Flag'](offset=32, bit=12)),
    (Field['Flag'](name="flags.badcentroid", doc="the centroid algorithm used to feed
centers to other algorithms failed"), Key['Flag'](offset=32, bit=13)),
    (Field['PointD'](name="centroid.sdss", doc="SDSS-algorithm centroid measurement",
units="pixels"), Key<PointD>(offset=72, nElements=2)),
    (Field['CovPointF'](name="centroid.sdss.err", doc="covariance matrix for centroid.
sdss", units="pixels^2"), Key<CovPointF>(offset=88, nElements=3)),
    (Field['Flag'](name="centroid.sdss.flags", doc="set if the centroid.sdss measureme
nt did not fully succeed"), Key['Flag'](offset=32, bit=14)),
    (Field['PointD'](name="centroid.naive", doc="unweighted 3x3 first moment centroi
d", units="pixels"), Key<PointD>(offset=104, nElements=2)),
    (Field['CovPointF'](name="centroid.naive.err", doc="covariance matrix for centroi
d.naive", units="pixels^2"), Key<CovPointF>(offset=120, nElements=3)),
    (Field['Flag'](name="centroid.naive.flags", doc="set if the centroid.naive measure
ment did not fully succeed"), Key['Flag'](offset=32, bit=15)),
    (Field['Flag'](name="flags.pixel.edge", doc="source is in region masked EDGE or NO
_DATA"), Key['Flag'](offset=32, bit=16)),
    (Field['Flag'](name="flags.pixel.interpolated.any", doc="source''s footprint inclu
des interpolated pixels"), Key['Flag'](offset=32, bit=17)),
    (Field['Flag'](name="flags.pixel.interpolated.center", doc="source''s center is cl
ose to interpolated pixels"), Key['Flag'](offset=32, bit=18)),
    (Field['Flag'](name="flags.pixel.saturated.any", doc="source''s footprint includes
saturated pixels"), Key['Flag'](offset=32, bit=19)),
    (Field['Flag'](name="flags.pixel.saturated.center", doc="source''s center is close
to saturated pixels"), Key['Flag'](offset=32, bit=20)),
    (Field['Flag'](name="flags.pixel.cr.any", doc="source''s footprint includes suspec
ted CR pixels"), Key['Flag'](offset=32, bit=21)),
    (Field['Flag'](name="flags.pixel.cr.center", doc="source''s center is close to sus
pected CR pixels"), Key['Flag'](offset=32, bit=22)),
    (Field['Flag'](name="flags.pixel.bad", doc="source is in region labeled BAD"), Key
['Flag'](offset=32, bit=23)),
    (Field['Flag'](name="flags.pixel.suspect.any", doc="source''s footprint includes s
uspect pixels"), Key['Flag'](offset=32, bit=24)),
```

```
    (Field['Flag'](name="flags.pixel.suspect.center", doc="source''s center is close t
o suspect pixels"), Key['Flag'](offset=32, bit=25)),
    (Field['PointD'](name="shape.hsm.bj.centroid", doc="PSF-corrected shear using Bern
stein & Jarvis (2002) method (centroid)"), Key<PointD>(offset=136, nElements=2)),
    (Field['MomentsD'](name="shape.hsm.bj.moments", doc="PSF-corrected shear using Ber
nstein & Jarvis (2002) method (uncorrected moments)"), Key<MomentsD>(offset=152, nElem
ents=3)),
    (Field['MomentsD'](name="shape.hsm.bj.psf", doc="PSF-corrected shear using Bernste
in & Jarvis (2002) method (PSF moments)"), Key<MomentsD>(offset=176, nElements=3)),
    (Field['D'](name="shape.hsm.bj.e1", doc="PSF-corrected shear using Bernstein & Jar
vis (2002) method (+ component of ellipticity)"), Key<D>(offset=200, nElements=1)),
    (Field['D'](name="shape.hsm.bj.e2", doc="PSF-corrected shear using Bernstein & Jar
vis (2002) method (x component of ellipticity)"), Key<D>(offset=208, nElements=1)),
    (Field['D'](name="shape.hsm.bj.err", doc="Uncertainty on e1 and e2 (assumed to be
the same)"), Key<D>(offset=216, nElements=1)),
    (Field['D'](name="shape.hsm.bj.sigma", doc="PSF-corrected shear using Bernstein &
Jarvis (2002) method (width)"), Key<D>(offset=224, nElements=1)),
    (Field['D'](name="shape.hsm.bj.resolution", doc="resolution factor (0=unresolved,
1=resolved)"), Key<D>(offset=232, nElements=1)),
    (Field['Flag'](name="shape.hsm.bj.flags", doc="set if measurement failed in any wa
y"), Key['Flag'](offset=32, bit=26)),
    (Field['PointD'](name="shape.hsm.ksb.centroid", doc="PSF-corrected shear using KSB
method (centroid)"), Key<PointD>(offset=240, nElements=2)),
    (Field['MomentsD'](name="shape.hsm.ksb.moments", doc="PSF-corrected shear using KS
B method (uncorrected moments)"), Key<MomentsD>(offset=256, nElements=3)),
    (Field['MomentsD'](name="shape.hsm.ksb.psf", doc="PSF-corrected shear using KSB me
thod (PSF moments)"), Key<MomentsD>(offset=280, nElements=3)),
    (Field['D'](name="shape.hsm.ksb.g1", doc="PSF-corrected shear using KSB method (+
component of estimated shear)"), Key<D>(offset=304, nElements=1)),
    (Field['D'](name="shape.hsm.ksb.g2", doc="PSF-corrected shear using KSB method (x
component of estimated shear)"), Key<D>(offset=312, nElements=1)),
    (Field['D'](name="shape.hsm.ksb.err", doc="Uncertainty on g1 and g2 (assumed to be
the same)"), Key<D>(offset=320, nElements=1)),
    (Field['D'](name="shape.hsm.ksb.sigma", doc="PSF-corrected shear using KSB method
(width)"), Key<D>(offset=328, nElements=1)),
    (Field['D'](name="shape.hsm.ksb.resolution", doc="resolution factor (0=unresolved,
1=resolved)"), Key<D>(offset=336, nElements=1)),
    (Field['Flag'](name="shape.hsm.ksb.flags", doc="set if measurement failed in any w
ay"), Key['Flag'](offset=32, bit=27)),
    (Field['PointD'](name="shape.hsm.linear.centroid", doc="PSF-corrected shear using
Hirata & Seljak (2003) ''linear'' method (centroid)"), Key<PointD>(offset=344, nElemen
ts=2)),
    (Field['MomentsD'](name="shape.hsm.linear.moments", doc="PSF-corrected shear using
Hirata & Seljak (2003) ''linear'' method (uncorrected moments)"), Key<MomentsD>(offset
=360, nElements=3)),
    (Field['MomentsD'](name="shape.hsm.linear.psf", doc="PSF-corrected shear using Hir
ata & Seljak (2003) ''linear'' method (PSF moments)"), Key<MomentsD>(offset=384, nElem
ents=3)),
    (Field['D'](name="shape.hsm.linear.e1", doc="PSF-corrected shear using Hirata & Se
ljak (2003) ''linear'' method (+ component of ellipticity)"), Key<D>(offset=408, nElem
ents=1)),
```

```
    (Field['D'](name="shape.hsm.linear.e2", doc="PSF-corrected shear using Hirata & Se
ljak (2003) ''linear'' method (x component of ellipticity)"), Key<D>(offset=416, nElem
ents=1)),
    (Field['D'](name="shape.hsm.linear.err", doc="Uncertainty on e1 and e2 (assumed to
be the same)"), Key<D>(offset=424, nElements=1)),
    (Field['D'](name="shape.hsm.linear.sigma", doc="PSF-corrected shear using Hirata &
Seljak (2003) ''linear'' method (width)"), Key<D>(offset=432, nElements=1)),
    (Field['D'](name="shape.hsm.linear.resolution", doc="resolution factor (0=unresolv
ed, 1=resolved)"), Key<D>(offset=440, nElements=1)),
    (Field['Flag'](name="shape.hsm.linear.flags", doc="set if measurement failed in an
y way"), Key['Flag'](offset=32, bit=28)),
    (Field['PointD'](name="shape.hsm.regauss.centroid", doc="PSF-corrected shear using
Hirata & Seljak (2003) ''regaussianization"), Key<PointD>(offset=448, nElements=2)),
    (Field['MomentsD'](name="shape.hsm.regauss.moments", doc="PSF-corrected shear usin
g Hirata & Seljak (2003) ''regaussianization"), Key<MomentsD>(offset=464, nElements=
3)),
    (Field['MomentsD'](name="shape.hsm.regauss.psf", doc="PSF-corrected shear using Hi
rata & Seljak (2003) ''regaussianization"), Key<MomentsD>(offset=488, nElements=3)),
    (Field['D'](name="shape.hsm.regauss.e1", doc="PSF-corrected shear using Hirata & S
eljak (2003) ''regaussianization"), Key<D>(offset=512, nElements=1)),
    (Field['D'](name="shape.hsm.regauss.e2", doc="PSF-corrected shear using Hirata & S
eljak (2003) ''regaussianization"), Key<D>(offset=520, nElements=1)),
    (Field['D'](name="shape.hsm.regauss.err", doc="Uncertainty on e1 and e2 (assumed t
o be the same)"), Key<D>(offset=528, nElements=1)),
    (Field['D'](name="shape.hsm.regauss.sigma", doc="PSF-corrected shear using Hirata
& Seljak (2003) ''regaussianization"), Key<D>(offset=536, nElements=1)),
    (Field['D'](name="shape.hsm.regauss.resolution", doc="resolution factor (0=unresol
ved, 1=resolved)"), Key<D>(offset=544, nElements=1)),
    (Field['Flag'](name="shape.hsm.regauss.flags", doc="set if measurement failed in a
ny way"), Key['Flag'](offset=32, bit=29)),
    (Field['PointD'](name="shape.hsm.shapelet.centroid", doc="PSF-corrected shear usin
g Hirata & Seljak (2003) ''shapelet'' method (centroid)"), Key<PointD>(offset=552, nEl
ements=2)),
    (Field['MomentsD'](name="shape.hsm.shapelet.moments", doc="PSF-corrected shear usi
ng Hirata & Seljak (2003) ''shapelet'' method (uncorrected moments)"), Key<MomentsD>(o
ffset=568, nElements=3)),
    (Field['MomentsD'](name="shape.hsm.shapelet.psf", doc="PSF-corrected shear using H
irata & Seljak (2003) ''shapelet'' method (PSF moments)"), Key<MomentsD>(offset=592, n
Elements=3)),
    (Field['D'](name="shape.hsm.shapelet.g1", doc="PSF-corrected shear using Hirata &
Seljak (2003) ''shapelet'' method (+ component of estimated shear)"), Key<D>(offset=61
6, nElements=1)),
    (Field['D'](name="shape.hsm.shapelet.g2", doc="PSF-corrected shear using Hirata &
Seljak (2003) ''shapelet'' method (x component of estimated shear)"), Key<D>(offset=62
4, nElements=1)),
    (Field['D'](name="shape.hsm.shapelet.err", doc="Uncertainty on g1 and g2 (assumed
to be the same)"), Key<D>(offset=632, nElements=1)),
    (Field['D'](name="shape.hsm.shapelet.sigma", doc="PSF-corrected shear using Hirata
& Seljak (2003) ''shapelet'' method (width)"), Key<D>(offset=640, nElements=1)),
    (Field['D'](name="shape.hsm.shapelet.resolution", doc="resolution factor (0=unreso
lved, 1=resolved)"), Key<D>(offset=648, nElements=1)),
```

```
    (Field['Flag'](name="shape.hsm.shapelet.flags", doc="set if measurement failed in
any way"), Key['Flag'](offset=32, bit=30)),
    (Field['MomentsD'](name="shape.sdss", doc="shape measured with SDSS adaptive momen
t algorithm", units="pixels^2"), Key<MomentsD>(offset=656, nElements=3)),
    (Field['CovMomentsF'](name="shape.sdss.err", doc="covariance matrix for shape.sds
s", units="pixels^4"), Key<CovMomentsF>(offset=680, nElements=6)),
    (Field['Flag'](name="shape.sdss.flags", doc="set if the shape.sdss measurement fai
led"), Key['Flag'](offset=32, bit=31)),
    (Field['PointD'](name="shape.sdss.centroid", doc="centroid measured with SDSS adap
tive moment shape algorithm", units="pixels"), Key<PointD>(offset=704, nElements=2)),
    (Field['CovPointF'](name="shape.sdss.centroid.err", doc="covariance matrix for sha
pe.sdss.centroid", units="pixels^2"), Key<CovPointF>(offset=720, nElements=3)),
    (Field['Flag'](name="shape.sdss.centroid.flags", doc="set if the shape.sdss.centro
id measurement did not fully succeed"), Key['Flag'](offset=32, bit=32)),
    (Field['Flag'](name="shape.sdss.flags.unweightedbad", doc="even the unweighted mom
ents were bad"), Key['Flag'](offset=32, bit=33)),
    (Field['Flag'](name="shape.sdss.flags.unweighted", doc="adaptive moments failed; f
all back to unweighted moments"), Key['Flag'](offset=32, bit=34)),
    (Field['Flag'](name="shape.sdss.flags.shift", doc="centroid shifted while estimati
ng adaptive moments"), Key['Flag'](offset=32, bit=35)),
    (Field['Flag'](name="shape.sdss.flags.maxiter", doc="too many iterations for adapt
ive moments"), Key['Flag'](offset=32, bit=36)),
    (Field['MomentsD'](name="shape.sdss.psf", doc="adaptive moments of the PSF model a
t the position of this object"), Key<MomentsD>(offset=736, nElements=3)),
    (Field['Flag'](name="shape.sdss.flags.psf", doc="failure in measuring PSF model sh
ape"), Key['Flag'](offset=32, bit=37)),
    (Field['ArrayD'](name="flux.aperture", doc="sum of pixels in apertures", units="d
n", size=10), Key<ArrayD>(offset=760, nElements=10)),
    (Field['ArrayD'](name="flux.aperture.err", doc="uncertainty for flux.aperture", un
its="dn", size=10), Key<ArrayD>(offset=840, nElements=10)),
    (Field['I'](name="flux.aperture.nProfile", doc="pixels", units="Number of points i
n radial profile successfully measured"), Key<I>(offset=920, nElements=1)),
    (Field['Flag'](name="flux.aperture.flags", doc="success flag for flux.aperture"),
Key['Flag'](offset=32, bit=38)),
    (Field['D'](name="flux.gaussian", doc="linear fit to an elliptical Gaussian with s
hape parameters set by adaptive moments", units="dn"), Key<D>(offset=928, nElements=
1)),
    (Field['D'](name="flux.gaussian.err", doc="uncertainty for flux.gaussian", units
="dn"), Key<D>(offset=936, nElements=1)),
    (Field['Flag'](name="flux.gaussian.flags", doc="set if the flux.gaussian measureme
nt failed"), Key['Flag'](offset=32, bit=39)),
    (Field['D'](name="flux.kron", doc="Kron photometry: photometry with aperture set t
o some multiple of <radius>determined within some multiple of the source size", units
="dn"), Key<D>(offset=944, nElements=1)),
    (Field['D'](name="flux.kron.err", doc="uncertainty for flux.kron", units="dn"), Ke
y<D>(offset=952, nElements=1)),
    (Field['Flag'](name="flux.kron.flags", doc="set if the flux.kron measurement faile
d"), Key['Flag'](offset=32, bit=40)),
    (Field['F'](name="flux.kron.radius", doc="Kron radius (sqrt(a*b))"), Key<F>(offset
=960, nElements=1)),
    (Field['F'](name="flux.kron.radiusForRadius", doc="Radius used to estimate <radius
```

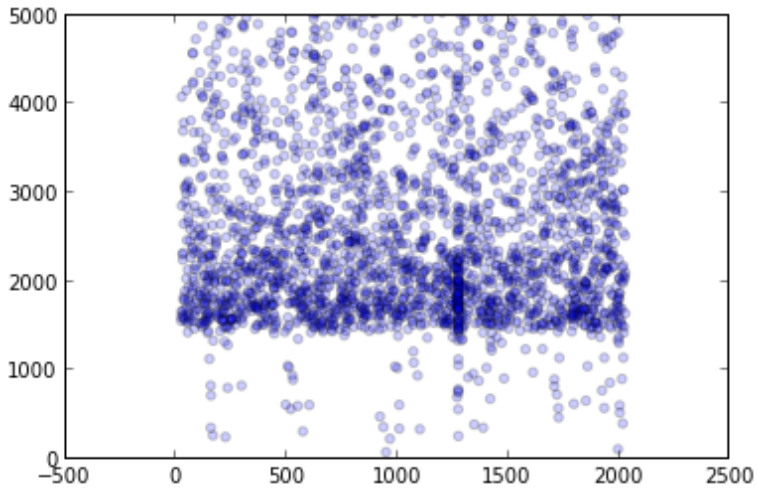
```
> (sqrt(a*b))"), Key<F>(offset=964, nElements=1)),
    (Field['Flag'](name="flux.kron.flags.radius", doc="Bad Kron radius (too close to e
dge to measure R_K)"), Key['Flag'](offset=32, bit=41)),
    (Field['Flag'](name="flux.kron.flags.smallRadius", doc="Measured Kron radius was s
maller than that of the PSF"), Key['Flag'](offset=32, bit=42)),
    (Field['D'](name="flux.naive", doc="simple sum over pixels in a circular apertur
e", units="dn"), Key<D>(offset=968, nElements=1)),
    (Field['D'](name="flux.naive.err", doc="uncertainty for flux.naive", units="dn"),
Key<D>(offset=976, nElements=1)),
    (Field['Flag'](name="flux.naive.flags", doc="set if the flux.naive measurement fai
led"), Key['Flag'](offset=32, bit=43)),
    (Field['D'](name="flux.psf", doc="flux measured by a fit to the PSF model", units
="dn"), Key<D>(offset=984, nElements=1)),
    (Field['D'](name="flux.psf.err", doc="uncertainty for flux.psf", units="dn"), Key<
D>(offset=992, nElements=1)),
    (Field['Flag'](name="flux.psf.flags", doc="set if the flux.psf measurement faile
d"), Key['Flag'](offset=32, bit=44)),
    (Field['D'](name="flux.sinc", doc="elliptical aperture photometry using sinc inter
polation", units="dn"), Key<D>(offset=1000, nElements=1)),
    (Field['D'](name="flux.sinc.err", doc="uncertainty for flux.sinc", units="dn"), Ke
y<D>(offset=1008, nElements=1)),
    (Field['Flag'](name="flux.sinc.flags", doc="set if the flux.sinc measurement faile
d"), Key['Flag'](offset=32, bit=45)),
    (Field['D'](name="flux.gaussian.apcorr", doc="aperture correction applied to flux.
gaussian"), Key<D>(offset=1016, nElements=1)),
    (Field['D'](name="flux.gaussian.apcorr.err", doc="error on aperture correction app
lied to flux.gaussian"), Key<D>(offset=1024, nElements=1)),
    (Field['Flag'](name="flux.gaussian.flags.apcorr", doc="set if aperture correction
lookup failed"), Key['Flag'](offset=32, bit=46)),
    (Field['D'](name="flux.kron.apcorr", doc="aperture correction applied to flux.kro
n"), Key<D>(offset=1032, nElements=1)),
    (Field['D'](name="flux.kron.apcorr.err", doc="error on aperture correction applied
to flux.kron"), Key<D>(offset=1040, nElements=1)),
    (Field['Flag'](name="flux.kron.flags.apcorr", doc="set if aperture correction look
up failed"), Key['Flag'](offset=32, bit=47)),
    (Field['D'](name="flux.psf.apcorr", doc="aperture correction applied to flux.ps
f"), Key<D>(offset=1048, nElements=1)),
    (Field['D'](name="flux.psf.apcorr.err", doc="error on aperture correction applied
to flux.psf"), Key<D>(offset=1056, nElements=1)),
    (Field['Flag'](name="flux.psf.flags.apcorr", doc="set if aperture correction looku
p failed"), Key['Flag'](offset=32, bit=48)),
    (Field['PointF'](name="focalplane", doc="Focal plane position"), Key<PointF>(offse
t=1064, nElements=2)),
    (Field['D'](name="jacobian", doc="Jacobian correction"), Key<D>(offset=1072, nElem
ents=1)),
    (Field['D'](name="classification.extendedness", doc="probability of being extende
d"), Key<D>(offset=1080, nElements=1)),
    (Field['Flag'](name="classification.exposure.photometric", doc="set if source was
used in photometric calibration"), Key['Flag'](offset=32, bit=49)),
    (Field['Flag'](name="classification.photometric", doc="set if source was used in p
hotometric calibration"), Key['Flag'](offset=32, bit=50)),
```

```
)
```

A catalog's schema is essentially a list of its columns (called Fields). In addition to its name and type, each field also typically has some documentation and units (if appropriate). Most of the time, you'll want to access the data column-by-columnn, which you can do by just calling `catalog.get()` with the name of the field. That returns a NumPy array, so you can use it directly for plotting. For instance, he's a plot of x position on the chip vs. PSF flux:

In [6]:
```
from matplotlib import pyplot
%matplotlib inline
pyplot.scatter(catalog.get("centroid.sdss.x"), catalog.get("flux.psf"), alpha=0.2)
pyplot.ylim(0, 5000)
```

Out[6]:
```
(0, 5000)
```



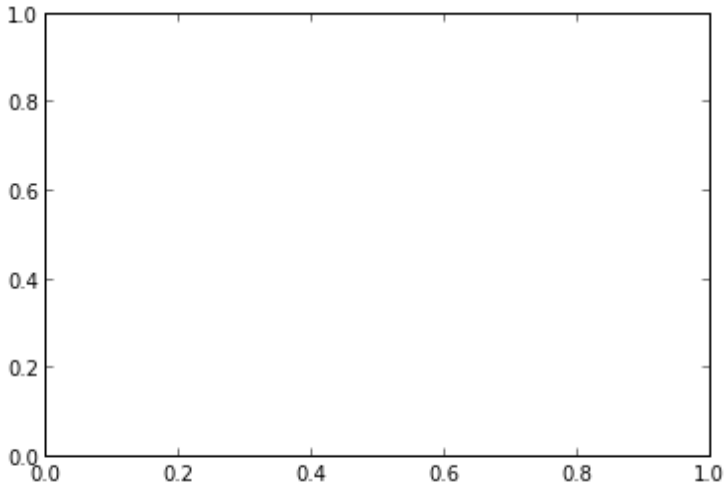
If you're paying very close attention, you may have noticed that while "`centroid.sdss`" was in the schema we printed, "`centroid.sdss.x`" was not. That's because "`centroid.sdss`" has type "`PointD`", which is a *compound field*. You can't actually get columns for an entire compound field, because there's no NumPy type for them, but you can get their implicit subfields, which for points are just "`x`" and "`y`". You can find information on the field types in the reference documentation: http://hsca.ipmu.jp/doxygen/latest/afw_table.html.

If we try to make a histogram of the fluxes, however, we'll run into problems:

In [7]:
```
pyplot.hist(catalog.get("flux.psf"), bins=50)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-9823b6a91187> in <module>()
----> 1 pyplot.hist(catalog.get("flux.psf"), bins=50)

/usr/lib/pymodules/python2.7/matplotlib/pyplot.pyc in hist(x, bins, range, normed, wei
ghts, cumulative, bottom, histtype, align, orientation, rwidth, log, color, label, hol
d, **kwargs)
   2339         ax.hold(hold)
   2340     try:
-> 2341         ret = ax.hist(x, bins, range, normed, weights, cumulative, bottom, his
ttype, align, orientation, rwidth, log, color, label, **kwargs)
   2342         draw_if_interactive()
```

```
   2343     finally:

/usr/lib/pymodules/python2.7/matplotlib/axes.pyc in hist(self, x, bins, range, normed,
weights, cumulative, bottom, histtype, align, orientation, rwidth, log, color, label,
 **kwargs)
   7732             # this will automatically overwrite bins,
   7733             # so that each histogram uses the same bins
-> 7734             m, bins = np.histogram(x[i], bins, weights=w[i], **hist_kwargs)
   7735             if normed:
   7736                 db = np.diff(bins)

/usr/lib/python2.7/dist-packages/numpy/lib/function_base.pyc in histogram(a, bins, ran
ge, normed, weights, density)
    156         if (mn > mx):
    157             raise AttributeError(
--> 158                 'max must be larger than min in range parameter.')
    159
    160     if not iterable(bins):

AttributeError: max must be larger than min in range parameter.
```



That's because many of the PSF flux measurements failed, and hence the array we get back has a lot of NaNs, and matplotlib's histogram doesn't like NaNs. To filter those out, we need to filter the catalog, and to do that, we want to look at `Flag` fields. `Flag` fields look just like booleans - when you get a column of them, they just return a bool array - but within the catalog each `Flag` value only takes up a single bit, so you don't need to worry about the fact that we have a lot of them. Most measurement algorithms have several flags, to indicate failure modes. Let's look at the flags for the `shape.sdss` algorithm:

In [8]:
```
for k, v in catalog.schema.extract("shape.sdss.flags*").iteritems():
    print "%s: %s" % (k, v.field.getDoc())
```

```
shape.sdss.flags.maxiter: too many iterations for adaptive moments
shape.sdss.flags.psf: failure in measuring PSF model shape
shape.sdss.flags: set if the shape.sdss measurement failed
shape.sdss.flags.unweighted: adaptive moments failed; fall back to unweighted moments
shape.sdss.flags.shift: centroid shifted while estimating adaptive moments
shape.sdss.flags.unweightedbad: even the unweighted moments were bad
```

The `extract` method is a way to get a view of just small piece of a `Schema`, using a glob pattern (it can also use regular expressions). It returns a `dict` of field names to `SchemaItem`s, which are just structs that hold a `Field` and a `Key`. The `Field` contains all the descriptive information, like the `getDoc()` method we use here. We'll return to `Key` objects later.

You can also call `extract` on a catalog, to get a `dict` of names and column arrays:

In [9]:
```
catalog.extract("flux.psf*")
```

Out[9]:
```
{'flux.psf': array([           nan,            nan,            nan, ...,  4244.1536945
2,
        2120.79321054,  1947.2929786 ]),
 'flux.psf.apcorr': array([ 0.97459748,  0.96734729,  0.96367793, ...,  0.98681612,
        0.98719534,  0.98712597]),
 'flux.psf.apcorr.err': array([ 0.0197787,  0.0197787,  0.0197787, ...,  0.0197787,
0.0197787,
        0.0197787]),
 'flux.psf.err': array([          nan,           nan,           nan, ...,  224.5112233
8,
        210.7598918 ,  209.6006962 ]),
 'flux.psf.flags': array([ True,  True,  True, ..., False, False, False], dtype=bool),
 'flux.psf.flags.apcorr': array([False, False, False, ..., False, False, False], dtype
=bool)}
```

You may have noticed that both `shape.sdss` and `flux.psf` have one or more fields starting with `*.flags`, with one `Flag` field named just `<algorithm>.flags`. That `Flag` field is essentially a combination of all the ones below it that represent a particular problem with the outputs (not all flags do), and an algorithm *should* never return NaN unless it also sets that "general failure" flag field. (Sometimes an algorithm will still produce a result, even though it does set the general failure flag, but this usually means the result shouldn't be trusted).

To get back to the problem of making a histogram of PSF fluxes, we now know we can use the "`flux.psf.flags`" field as a filter. To apply that filter, we simply index the catalog with a bool array that's `True` for all the objects we want to keep (just as we would index a NumPy array). Because the `Flag` field is actually `False` for all the objects, we want to keep, we need to invert it first:

In [10]:
```
import numpy
good = catalog[numpy.logical_not(catalog.get("flux.psf.flags"))]
pyplot.hist(good.get("flux.psf"), bins=50)
```

```
---------------------------------------------------------------------------
LsstCppException                          Traceback (most recent call last)
<ipython-input-10-5d403603af30> in <module>()
      1 import numpy
      2 good = catalog[numpy.logical_not(catalog.get("flux.psf.flags"))]
----> 3 pyplot.hist(good.get("flux.psf"), bins=50)

/mnt/data/jbosch/HSC/master/afw/python/lsst/afw/table/tableLib.pyc in get(self, k)
   8292     def get(self, k):
   8293         """Synonym for self[k]; provided for consistency with C++ interfac
e."""
-> 8294         return self[k]
   8295
   8296     def cast(self, type_, deep=False):
```

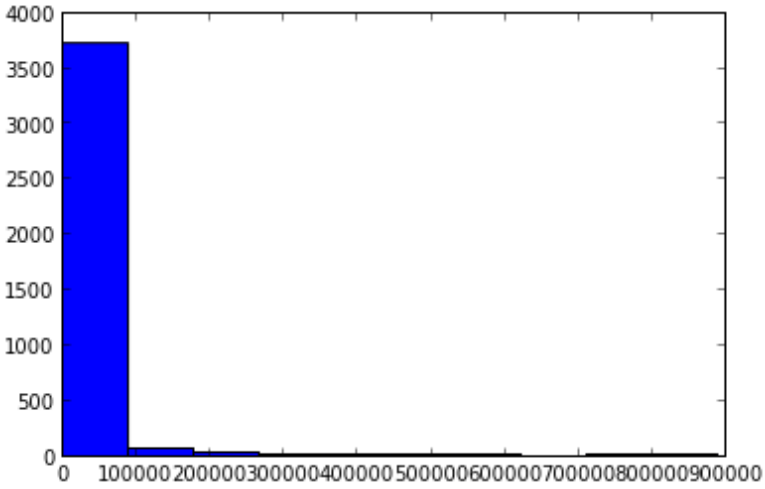
```
/mnt/data/jbosch/HSC/master/afw/python/lsst/afw/table/tableLib.pyc in __getitem__(sel
f, k)
   8224             return _tableLib._SourceCatalogBase___getitem__(self, k)
   8225         except TypeError:
-> 8226             return self.columns[k]
   8227 
   8228 

/mnt/data/jbosch/HSC/master/afw/python/lsst/afw/table/tableLib.pyc in __getattribute__
(self, name)
   8317         # We have to use __getattribute__ because SWIG overrides __getattr__.
   8318         try:
-> 8319             return object.__getattribute__(self, name)
   8320         except AttributeError:
   8321             # self._columns is created the when self.columns is accessed -

/mnt/data/jbosch/HSC/master/afw/python/lsst/afw/table/tableLib.pyc in __getColumns(sel
f)
   8282     def __getColumns(self):
   8283         if not hasattr(self, "_columns") or self._columns is None:
-> 8284             self._columns = self.getColumnView()
   8285         return self._columns
   8286     columns = property(__getColumns, doc="a column view of the catalog")

/mnt/data/jbosch/HSC/master/afw/python/lsst/afw/table/tableLib.pyc in getColumnView(se
lf)
   8168     def getColumnView(self):
   8169         """getColumnView(self) -> ColumnView"""
-> 8170         val = _tableLib._SourceCatalogBase_getColumnView(self)
   8171         self._columns = val
   8172 

LsstCppException: 0: lsst::pex::exceptions::RuntimeErrorException thrown at include/ls
st/afw/table/BaseColumnView.h:205 in static lsst::afw::table::BaseColumnView lsst::af
w::table::BaseColumnView::make(const boost::shared_ptr<lsst::afw::table::BaseTable>&,
 InputIterator, InputIterator) [with InputIterator = lsst::afw::table::CatalogIterator
<__gnu_cxx::__normal_iterator<const boost::shared_ptr<lsst::afw::table::SourceRecord>
*, std::vector<boost::shared_ptr<lsst::afw::table::SourceRecord>, std::allocator<boos
t::shared_ptr<lsst::afw::table::SourceRecord> > > > >]
0: Message: Record data is not contiguous in memory.
```

The reason things didn't work this time is that when we slice the catalog, the new catalog we get back is a view into the old one. And that means that the rows (i.e. records) of the catalog are not contiguous in memory. The columns we get back as NumPy arrays are also views into the catalog, and NumPy's data model requires that those be contiguous. So to get columns from a catalog, we need that catalog to be contiguous. Happily, that's easy - we just deep-copy that catalog after we slice it:

In [11]:
```
good = catalog[numpy.logical_not(catalog.get("flux.psf.flags"))].copy(deep=True)
pyplot.hist(good.get("flux.psf"), bins=10)
```

Out[11]:
```
(array([3730,   68,   32,   19,    6,    7,   15,    3,    6,   12]),
 array([  7.47075200e+01,   8.88406181e+04,   1.77606529e+05,
```

```
        2.66372439e+05,   3.55138350e+05,   4.43904260e+05,
        5.32670171e+05,   6.21436082e+05,   7.10201992e+05,
        7.98967903e+05,   8.87733813e+05]),
<a list of 10 Patch objects>)
```



Records and Keys

We can also iterate over catalogs row-by-row instead instead of column-by-column. Here's a slower way of getting an array of unflagged PSF fluxes:

In [12]:
```
fluxes = []
for record in catalog:
    if not record.get("flux.psf.flags"):
        fluxes.append("flux.psf")
```

This is slow for two reasons: - When we get columns, we loop over all the rows in C++, which is much faster than looping over rows in Python. - Looking up a field by name is actually quite slow, and while we only do that once for all rows when we get an entire column, in the code above we do it separately for every row.

Sometimes you need to loop over every row in Python anyway, in order to perform some computation. When you do that, you can avoid the second problem by using `Key` objects:

In [13]:
```
flagKey = catalog.schema.find("flux.psf.flags").key
fluxKey = catalog.schema.find("flux.psf").key
fluxes = []
for record in catalog:
    if not record.get(flagKey):
        fluxes.append(fluxKey)
```

Using a `Key` object to get a value from a record is *much* faster than using a string, and you can use a `Key` anywhere you could use a string in field lookup (including columns, though usually it's no faster there, because you only use the `Key` once).

You can also get individual records that correspond to a particular source by their position in the catalog:

In [14]:
```
record = catalog[52]
```

Or by searching by Source ID (note that the ID is not the same as the position in the catalog!):

In [15]:
```
record = catalog.find(1063605701181802)
```

One thing you can do with records that you can't do with full catalogs is get compound fields. For instance, now we can get the full "centroid.sdss" field, as an lsst.afw.geom.Point2D:

In [16]:
```
point = record.get("centroid.sdss")
print point.getX(), point.getY()
```

```
436.0 6.0
```

Slots

Instead of using strings or keys, you can also get many fields using a system called "slots". Not only are these easy to use, they make it so you do not have to know the name of the algorithm we used to produce a measurement in order to get its result.

For example, we can also get the centroid like this:

In [17]:
```
point = record.getCentroid()
```

This is the same point we had before, because getCentroid is just an alias to "centroid.sdss". When we run the pipeline, we can configure which of several centroid algorithms is used by getCentroid. Because it returns a complex object, you cannot use getCentroid on catalogs, but you can use getX() and getY() on both catalogs and records to get the coordinates of getCentroid() separately:

In [18]:
```
assert record.getX() == record.get("centroid.sdss.x")
assert (catalog.getX() == catalog.get("centroid.sdss.x")).all()
```

In addition to the centroid, we have the following slots: - getShape() (also getIxx(), getIyy(), getIxy()) - moments of the object, usually an alias to "shape.sdss". - getPsfFlux() - a flux using a fit of the PSF model, usually an alias to "flux.psf". - getApFlux() - an aperture flux at a particular radius considered useful for the pipeline - getModelFlux() - a flux using some sort of model fit. In the coadd processing, this is cmodel.flux by default, but it is flux.gaussian by default on single frames, because the CModel flux is very slow. - getInstFlux() - means "instrumental flux", but it mostly exists for historical reasons

All of these slots also have uncertainty and flag getters (e.g. getPsfFluxErr()->"flux.psf.err", getPsfFluxFlag()->"flux.psf.flags").

When you can use one of these slots, it is probably better than using a string name, because it will keep your code from breaking if we change the names of algorithms in the future (it is much less likely that the slots will change), or decide to use a different algorithm in one of the slots (for instance, centroid.sdss is our best centroid measurement right now, but we may have a better one in the future).

Modifying Catalogs

Because columns are views into the catalog, not copies, if you modify them using the usual NumPy commands, you modify the catalog itself:

In [19]:
```
psfFlux = catalog.getPsfFlux()
psfFlux[2:4] = 0.0
```

An important exception is `Flag` fields: because these are stored as individual bits, getting a Flag column *does* make a copy, so modifying them does not modify the catalog. To modify a Flag, you have to use a record object.

To modify records, you use the `set` method:

In [20]:
```
record.set("flux.psf.flags", True)
```

If you want to add a record to a catalog, use `addNew()`:

In [21]:
```
newRecord = catalog.addNew()
```

You can then modify that record using `set`, but be warned: adding a record can break the "contiguousness" of the catalog's memory, making it impossible to get columns from the catalog (which you can fix by making a deep copy, as before). It won't always break the contiguousness - it depends on whether the catalog needs to allocate more memory to make space for the new record. There are many options to control the memory management of a catalog object, but that is beyond the scope of this tutorial.

To avoid doing that now (and breaking the examples below), we'll delete the record we just created:

In [22]:
```
del catalog[-1]
```

You can also add a single existing record to a catalog with `record.append`, or combine multiple catalogs together `record.extend`. We won't cover those further today, except to mention that all records in a catalog must have the same Schema, so you must make sure this is the case before adding more records to a catalog.

Adding columns to a catalog is much more difficult. In fact, the memory model of the catalog makes this impossible - instead, you have to create a new catalog with both the new columns and the old columns in the Schema, and then copy the columns from the old catalog to the new catalog. There is an object, `lsst.afw.table.SchemaMapper`, to help with this, and some other documentation exists to show how to use it for this purpose: http://hsca.ipmu.jp:8080/question/289/how-do-i-add-columns-to-an-existing-catalog/.

Getting Calibrated Measurements

Fluxes to Magnitudes

Usually when looking at flux measurements, we actually want to plot magnitudes instead of raw fluxes. To convert between them, we use a `Calib` object, which we can get from the `exposure` object we loaded back at the beginning:

In [23]:
```
calib = exposure.getCalib()
calib.setThrowOnNegativeFlux(False)  # don't raise an exception when we encounter a ne
gative or NaN flux
psfMag = calib.getMagnitude(catalog.getPsfFlux())
```

A `Calib` can also convert flux errors to magnitude errors at the same time:

In [24]:
```
psfMag, psfMagErr = calib.getMagnitude(catalog.getPsfFlux(), catalog.getPsfFluxErr())
```

Note that you can also use `calib.getMagnitude(...)` on a record object, where it will of course return single values instead of arrays.

Positions

If you want to transform (x, y) point measurements to (ra, dec) measurements, you probably don't need to do anything - for convenience, records and catalogs already have `getRA()` and `getDec()` methods, and records have a `getCoord()` method as well. But let's imagine that the `Wcs` object on our `exposure` is better than it was during the measurement, and we want to transform the points again. Here's how to do that:

In [25]:
```
wcs = exposure.getWcs()
coord = wcs.pixelToSky(record.getCentroid())
print coord
```

```
(2.62715 rad, 0.0373167 rad)
```

This sort of calibration can only be done on one record at a time, and you can see that the `pixelToSky(...)` method returns a `lsst.afw.coord.Coord` object, not a `Point2D`. A `Coord` knows what coordinate system it is in (ICRS vs. galactic, etc.), and it lets you convert between different angle units, or different coordinate systems:

In [26]:
```
print coord.getPosition(lsst.afw.geom.degrees)
print coord.toEcliptic().getPosition(lsst.afw.geom.degrees)
```

```
(150.52, 2.1381)
(151.83, -9.2859)
```

Ellipses

Converting shapes (ellipses) is a little trickier. We need to start by approximating the `Wcs` at the position of the object as an affine transformation:

In [27]:
```
affine = wcs.linearizePixelToSky(record.getCoord(), lsst.afw.geom.arcseconds)
moments = record.getShape().transform(affine.getLinear())
print moments
```

```
(ixx=0.396268212665, iyy=0.17668954635, ixy=0.016512118668)
```

Even though the parameters are still called (ixx, iyy, ixy), these are actually in arcseconds^2, and the parameters are defined with (x=ra, y=dec). We could have used `degrees`, `radians`, or `arcminutes` instead. We can also convert this ellipse into the (a, b, theta) parametrization, or an (e1, e2, r) parametrization as well. We'll do this after converting to celestial coordinates, but we could have done it before:

In [28]:
```
axes = lsst.afw.geom.ellipses.Axes(moments)
print axes.getA(), axes.getB(), axes.getTheta()
```

```
0.630478362678 0.418873242414 0.0746396838386
```

In [29]:
```
separable = lsst.afw.geom.ellipses.SeparableDistortionDeterminantRadius(moments)
print separable.getE1(), separable.getE2(), separable.getRadius()
```

```
0.38323709359 0.0576381710803 0.513897378906
```

There are many other `Separable` classes in `lsst.afw.geom.ellipses`, as there are many definitions of ellipticity and radius.

Mosaic Calibrations

If you have run the `mosaic` pipeline to calibrate many visits together, improved calibrates are saved on disk, but these are *not* immediately applied to the "`calexp`" and "`src`" datasets. This means that when you call `exposure.getWcs()`, you get the original single-frame-astrometry WCS, not the improved mosaic WCS. Reading and applying the improved astrometry and photometry is difficult, so we have written some utilities to do it for you.

The easiest way to use these utilities is to run the `calibrateCatalog.py` and `catalogExposure.py` command-line scripts. These create new butler datasets, "`calibrated_src`" and "`calibrated_exp`". If you simply load those instead of "`src`" and "`calexp`", you will get the improved calibrations automatically.