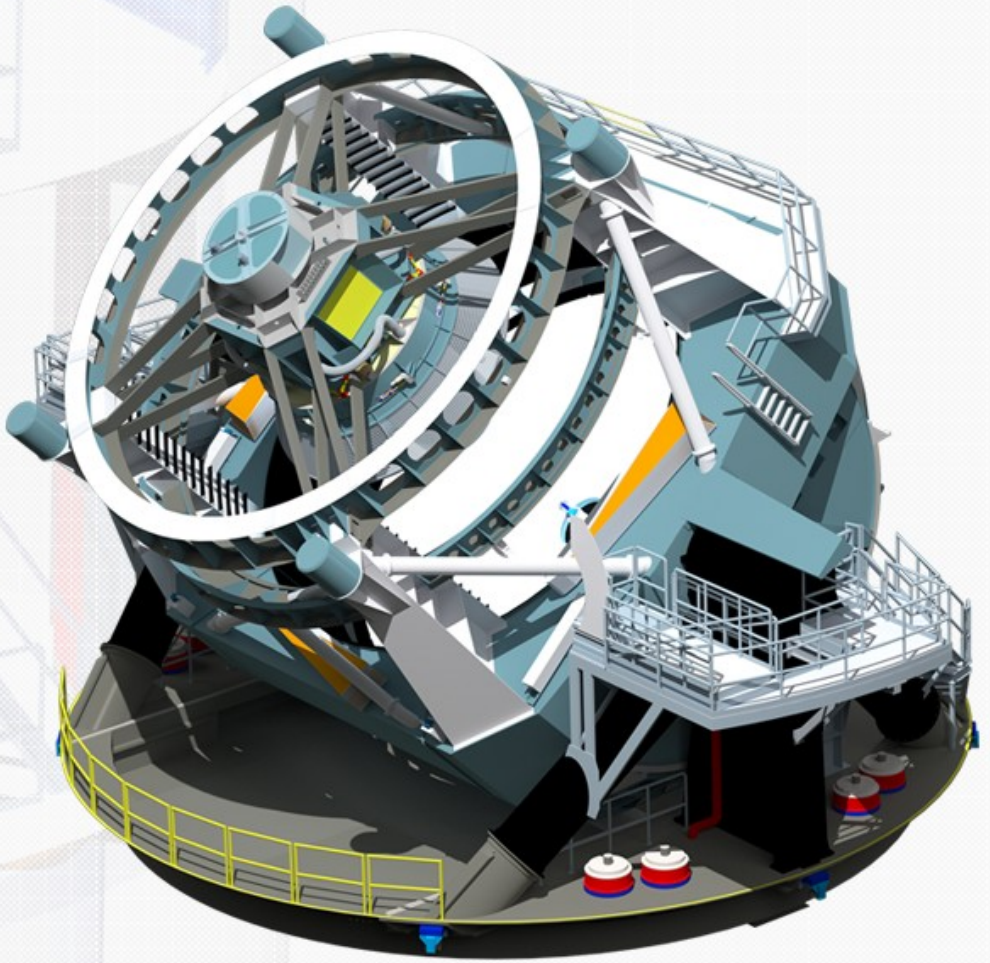
Source Measurement & Tables

Jim Bosch

October 7, 2015



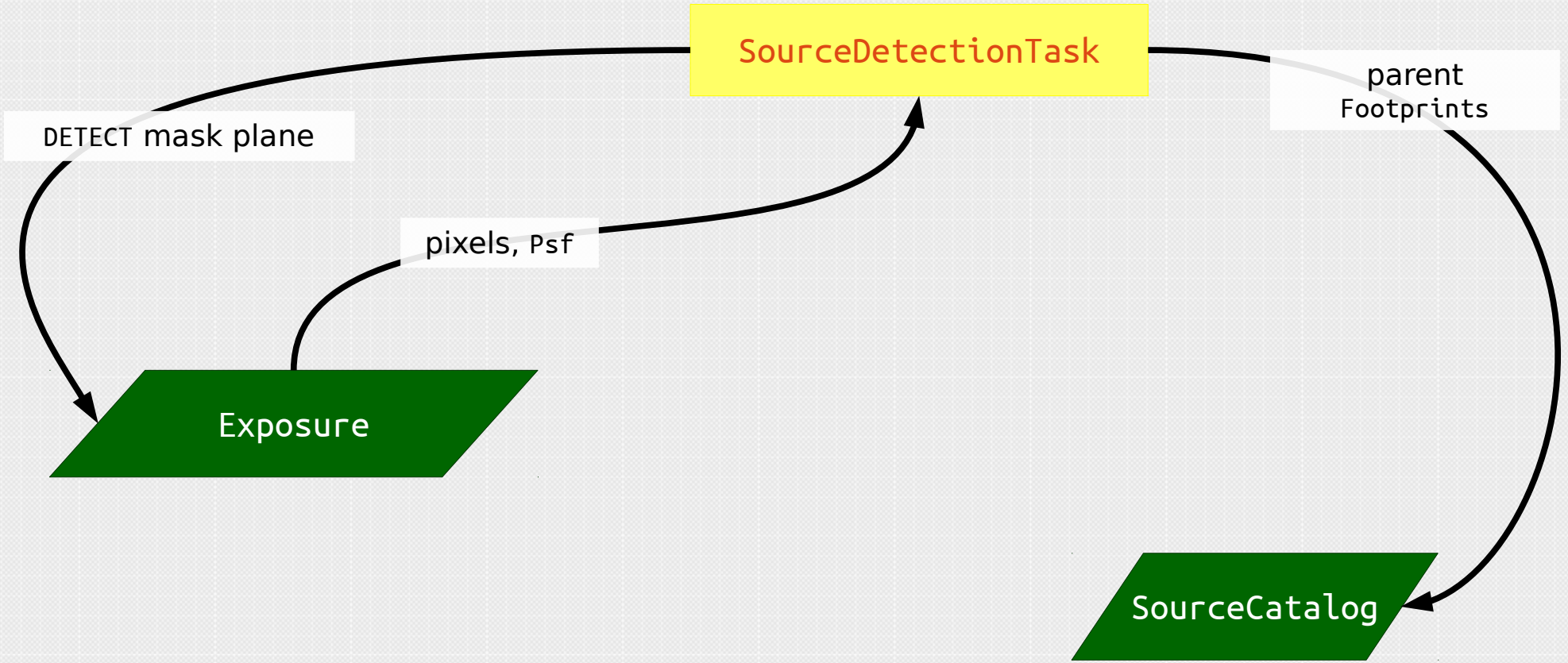
Outline

- **Measurement**
  - Detection, Deblending, and Measurement
  - Inside Measurement
  - Using `SingleFrameMeasurementTask`
  - Writing a `SingleFramePlugin`
    - Error Handling
    - Unit Transformation
    - C++ Algorithms
    - Utility Classes
  - Forced Measurement
- **Tables**
  - Records, Tables, and Catalogs
  - Catalog Indexing
  - `SchemaMapper`
  - Flag Fields
  - Slots and Aliases
  - FunctorKeys

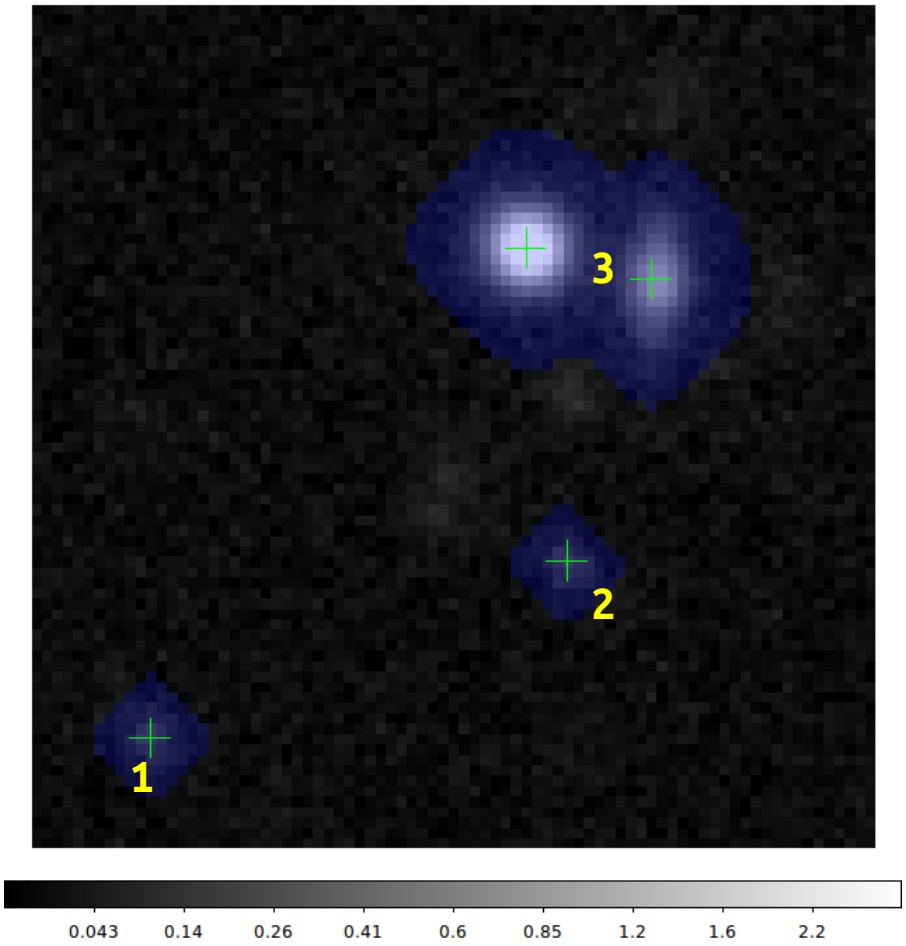
Adding a Column with SchemaMapper

```
# Load the original catalog from disk
catI = lsst.afw.table.SourceCatalog.readFits("forcedI.fits")

# Create a SchemaMapper that maps from the given Schema to a new one
# it will create.  This does not set up any mappings.
mapper = lsst.afw.table.SchemaMapper(catI.schema)

# Add everything from the given Schema to output Schema, creating mappings
# as we go (that's what 'True') does.
# addMinimalSchema can only be called before creating any output fields;
# by putting the new fields at the beginning, we ensure Keys from the old
# Schema work with the new one.
mapper.addMinimalSchema(catI.schema, True)

# Add a new field to the output schema.
mapper.editOutputSchema().addField("new_field", type=int,
                                   doc="A new field added in a SchemaMapper")

# Create an empty catalog with the SchemaMapper's output schema.
newCat = lsst.afw.table.SourceCatalog(mapper.getOutputSchema())

# Add all records from the old catalog to the new one, using the SchemaMapper
# to copy values.
newCat.extend(catI, mapper=mapper)
```

Detect, Deblend, Measure

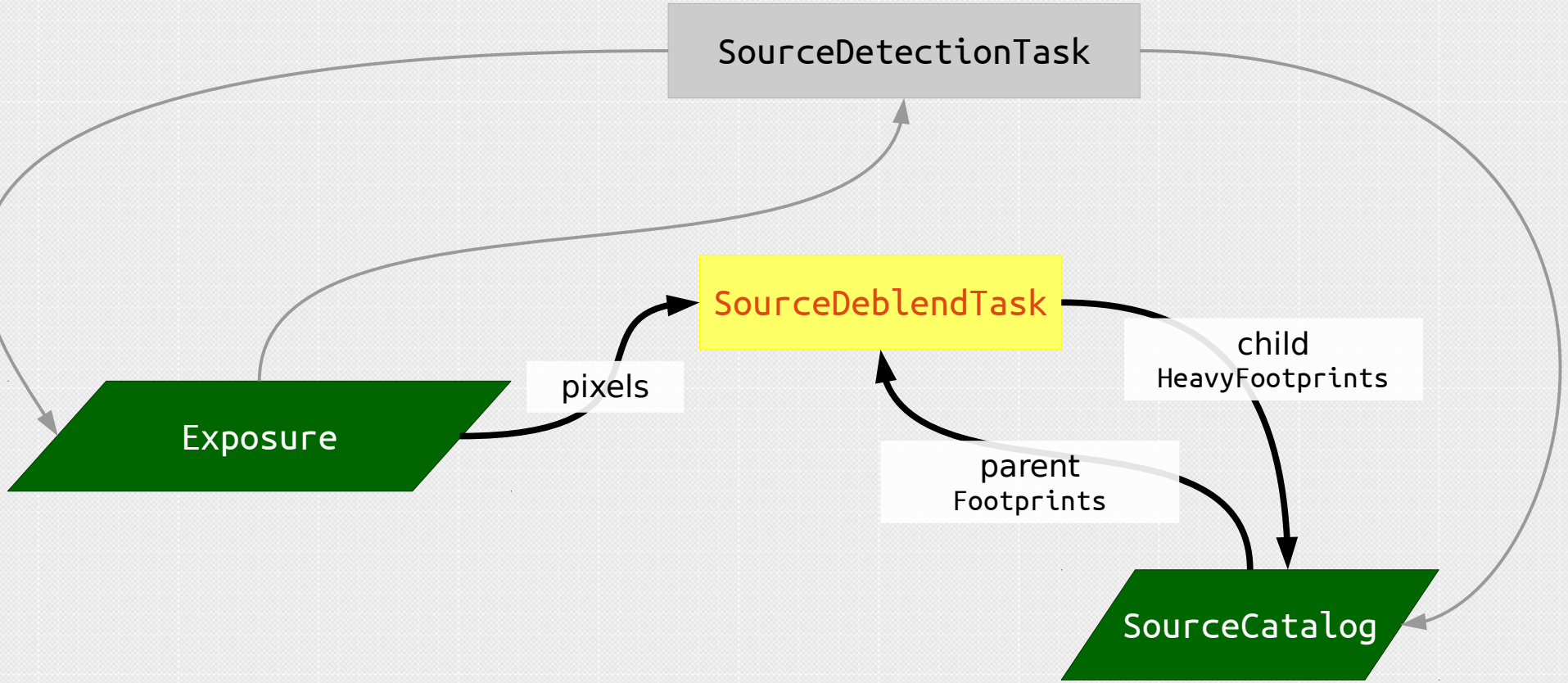


Detect, Deblend, Measure

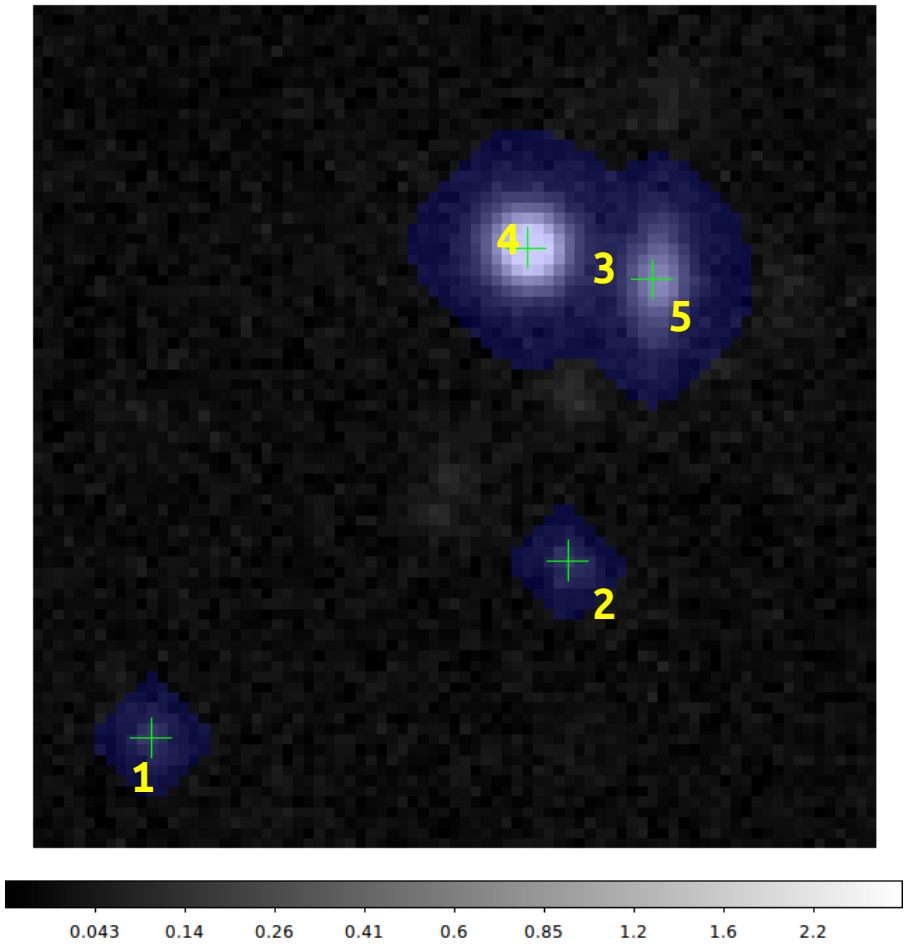


id	parent	nchild	(measurements)	(footprint)
1	0	0		regular
2	0	0		regular
3	0	0		regular

Detect, ***Deblend***, Measure



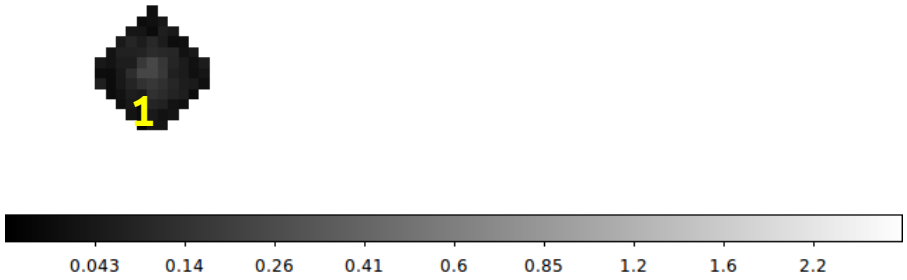
Detect, Deblend, Measure



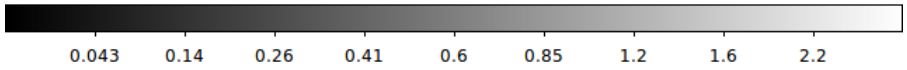
id	parent	nchild	(measurements)	(footprint)
1	0	0		regular
2	0	0		regular
3	0	2		regular
4	3	0		heavy
5	3	0		heavy

Detect, Deblend, Measure

id	parent	nchild	(measurements)	(footprint)
1	0	0		regular
2	0	0		regular
3	0	2		regular
4	3	0		heavy
5	3	0		heavy

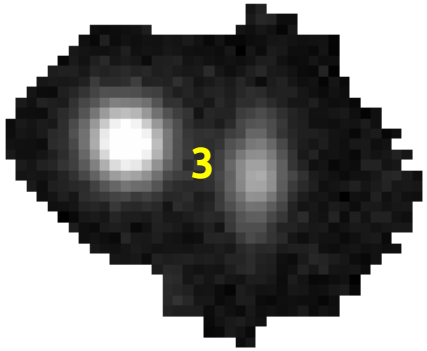


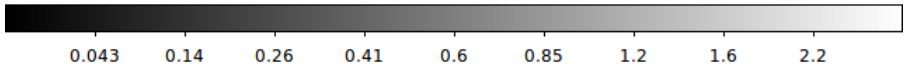
Detect, Deblend, Measure



id	parent	nchild	(measurements)	(footprint)
1	0	0		regular
2	0	0		regular
3	0	2		regular
4	3	0		heavy
5	3	0		heavy

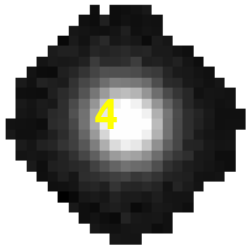
Detect, Deblend, Measure

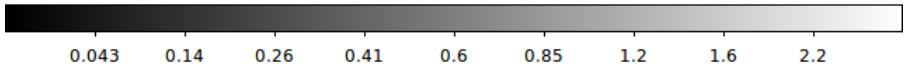




id	parent	nchild	(measurements)	(footprint)
1	0	0		regular
2	0	0		regular
3	0	2		regular
4	3	0		heavy
5	3	0		heavy

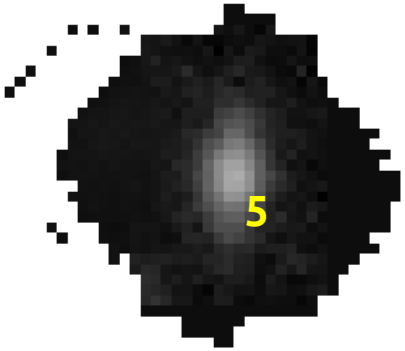
Detect, Deblend, Measure

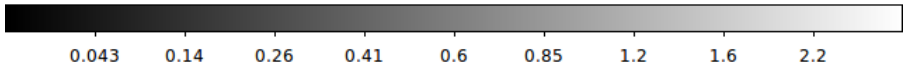




id	parent	nchild	(measurements)	(footprint)
1	0	0		regular
2	0	0		regular
3	0	2		regular
4	3	0		heavy
5	3	0		heavy

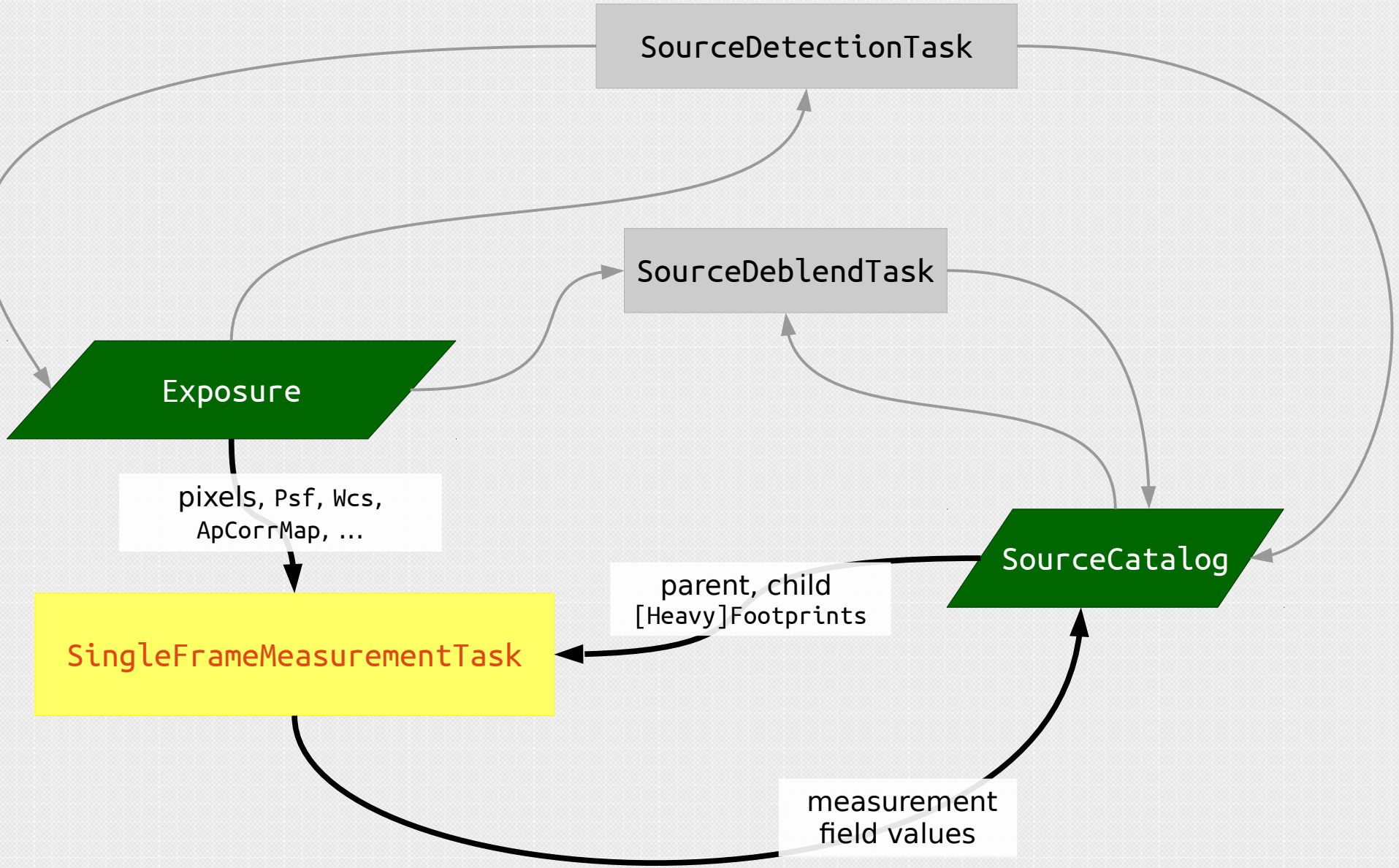
Detect, Deblend, Measure



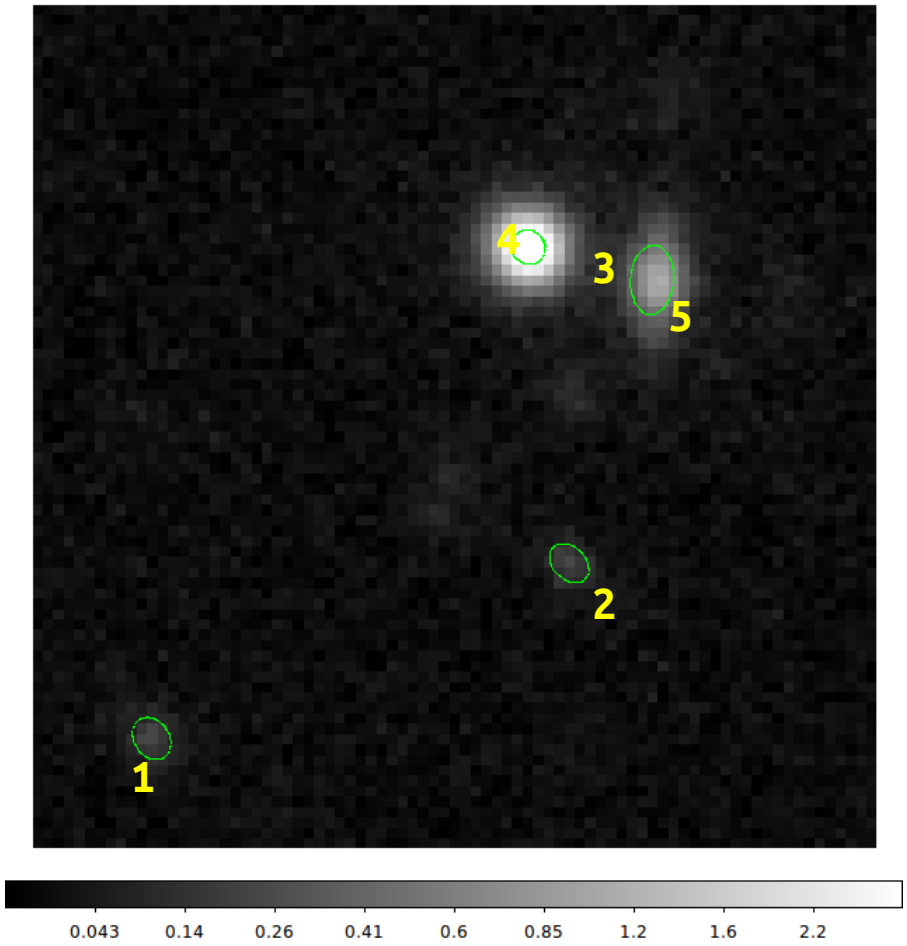


id	parent	nchild	(measurements)	(footprint)
1	0	0		regular
2	0	0		regular
3	0	2		regular
4	3	0		heavy
5	3	0		heavy

Detect, Deblend, **Measure**

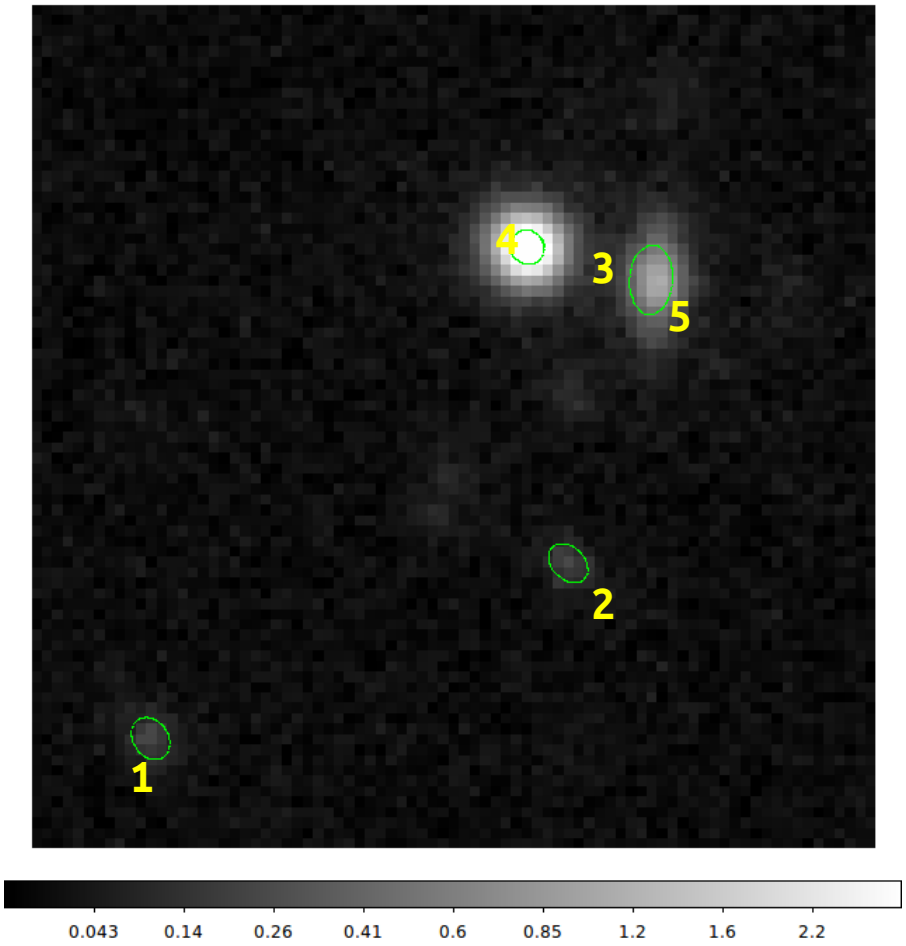


Detect, Deblend, ***Measure***



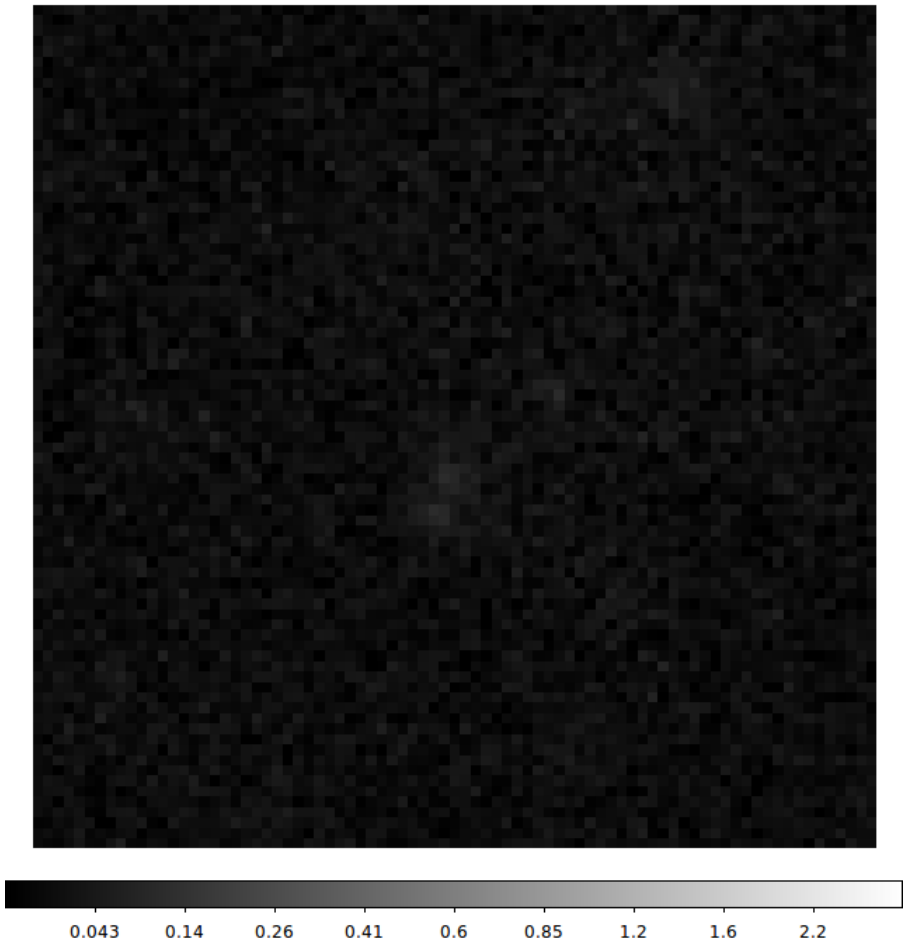
id	parent	nchild	(measurements)	(footprint)
1	0	0	(filled)	regular
2	0	0	(filled)	regular
3	0	2	(filled)	regular
4	3	0	(filled)	heavy
5	3	0	(filled)	heavy

Inside Measurement




```
replace all detections with noise

for record in catalog:
   restore pixels from HeavyFootprint
   run plugins
   re-replace pixels with noise

apply aperture corrections
```

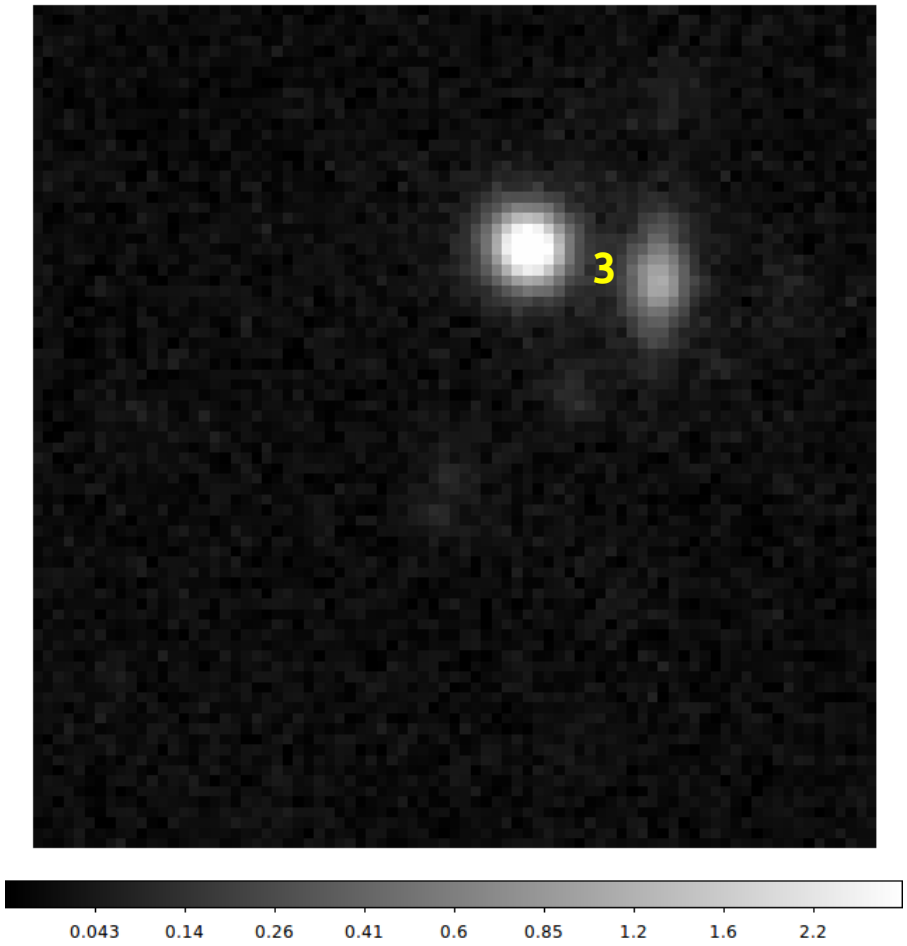
Inside Measurement



replace all detections with noise

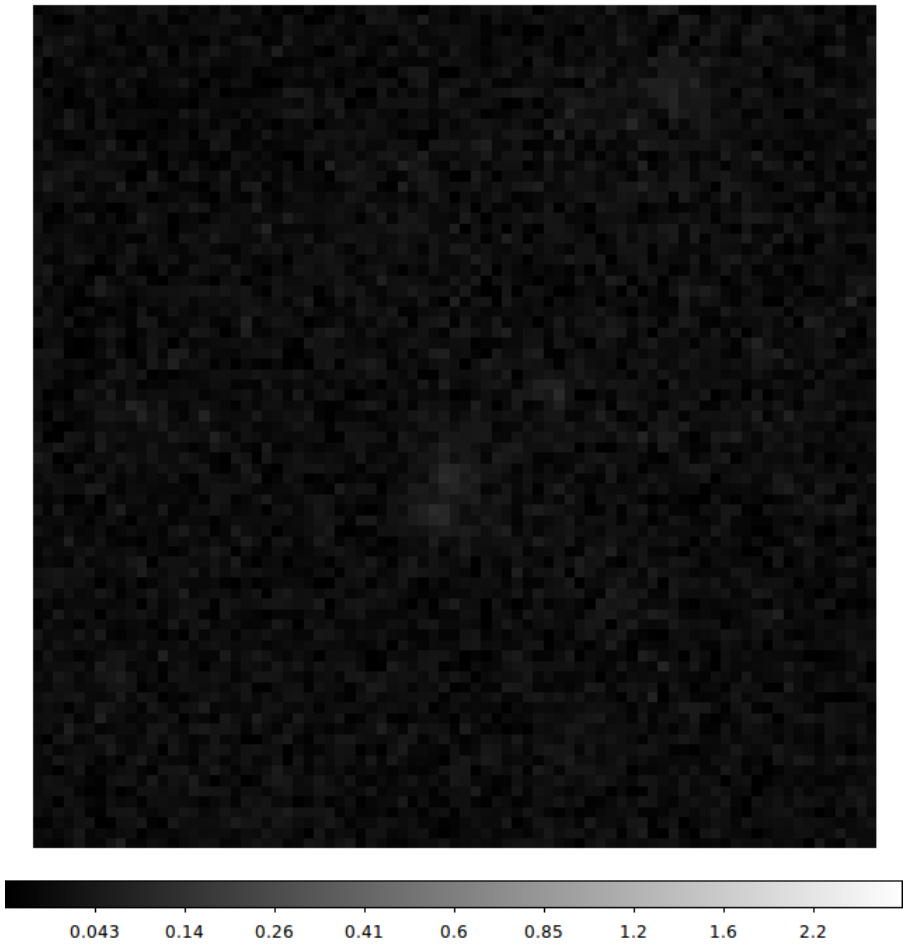
```
for record in catalog:
   restore pixels from HeavyFootprint
   run plugins
   re-replace pixels with noise

apply aperture corrections
```

Inside Measurement



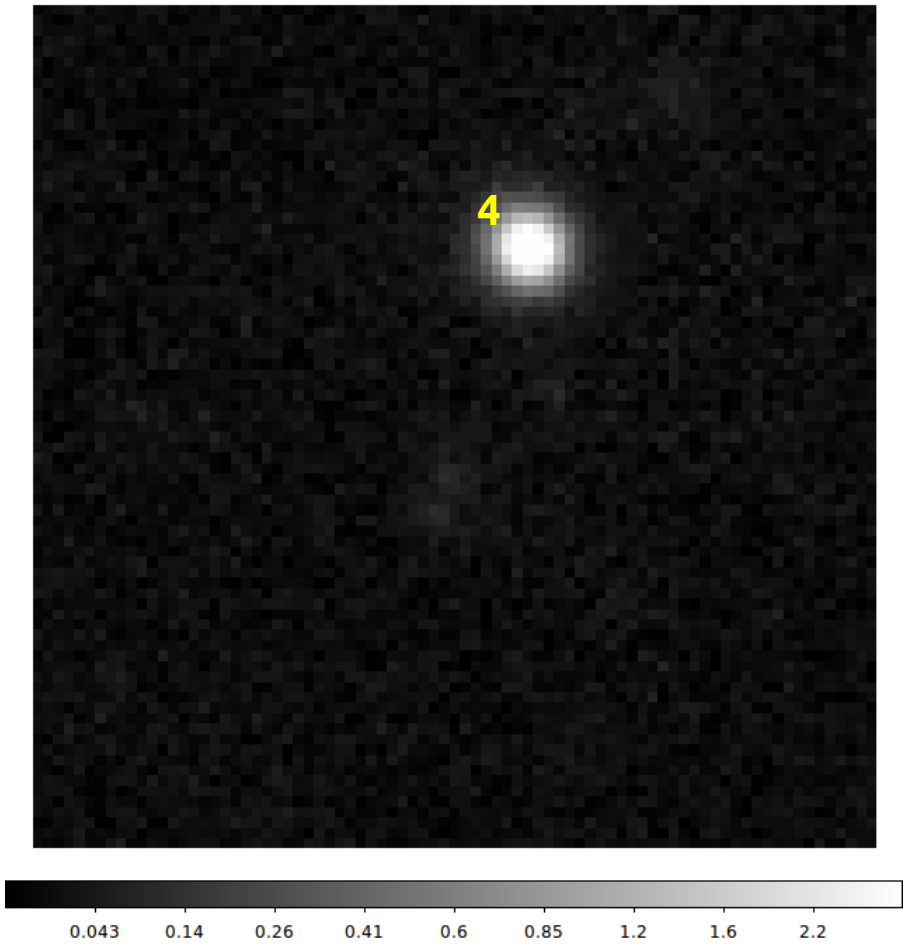

```
replace all detections with noise

for record in catalog:
   restore pixels from HeavyFootprint
   run plugins
   re-replace pixels with noise

apply aperture corrections
```

Inside Measurement



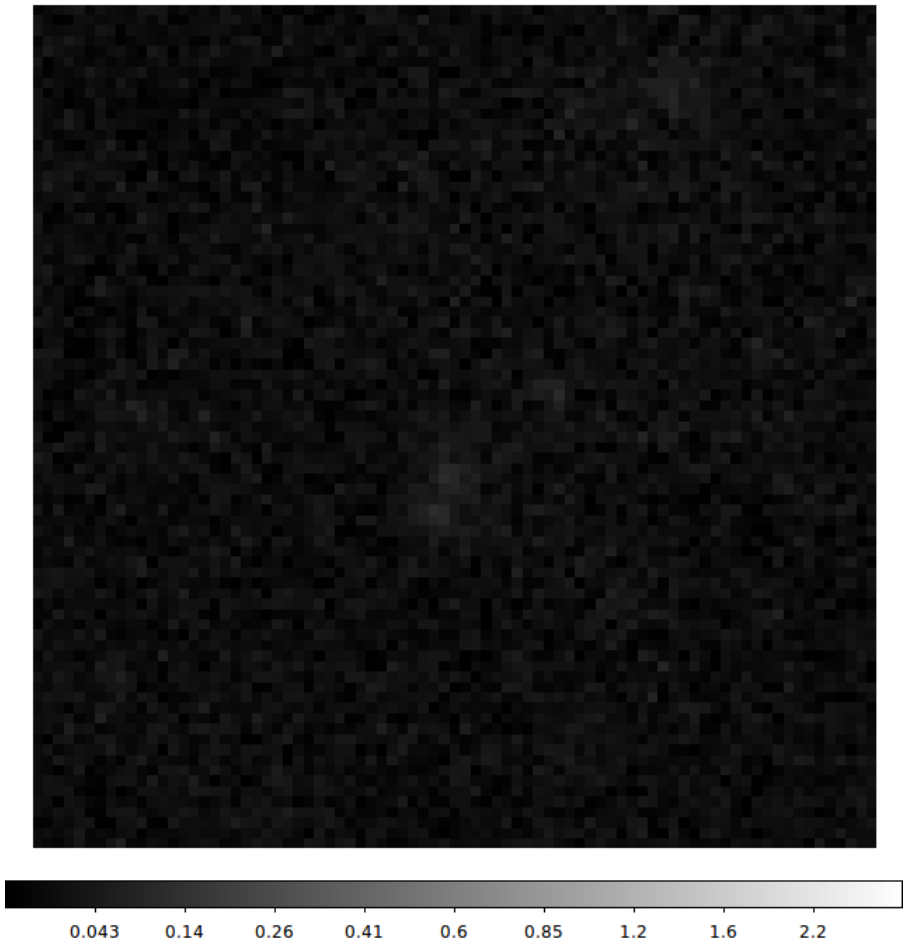

```
replace all detections with noise

for record in catalog:
   restore pixels from HeavyFootprint
   run plugins
   re-replace pixels with noise

apply aperture corrections
```

Inside Measurement




```
replace all detections with noise

for record in catalog:
   restore pixels from HeavyFootprint
   run plugins
   re-replace pixels with noise

apply aperture corrections
```

Inside Measurement




```
replace all detections with noise

for record in catalog:
   restore pixels from HeavyFootprint
   run plugins
   re-replace pixels with noise

apply aperture corrections
```

Before Initializing `SingleFrameMeasurementTask`

```
# We have to initialize all tasks before using any of them:
# multiple tasks will write to the same Schema, and we can't create an output
# catalog until we've finished defining that Schema.

# Start with a minimal schema - only the fields all SourceCatalogs need
schema = lsst.afw.table.SourceTable.makeMinimalSchema()

# Customize the detection task a bit (optional)
detectConfig = lsst.meas.algorithms.SourceDetectionConfig()
detectConfig.returnOriginalFootprints = False   # should be the default
detectConfig.thresholdValue = 10                # only 10-sigma detections

# Create the detection task.  We pass the schema so the task can declare a few flag fields
detectTask = lsst.meas.algorithms.SourceDetectionTask(config=detectConfig, schema=schema)

# Create a task for deblending (optional, but almost always a good idea).
# Again, the task defines a few flag fields it will later fill.
deblendTask = lsst.meas.deblender.SourceDeblendTask(schema=schema)
```

Configuring SingleFrameMeasurementTask

```
measureConfig = lsst.meas.base.SingleFrameMeasurementConfig()

# Modify the set of active plugins ('.names' behaves like a Python set)
measureConfig.plugins.names.remove("base_GaussianCentroid")

# Enable some plugins - import the Python module first to make them available
measureConfig.plugins.names |= ["modelfit_ShapeletPsfApprox", "modelfit_CModel"]

# Change which plugin's output we "bless" as the "Model Flux"
measureConfig.slots.modelFlux = "modelfit_CModel"

# Modify the internal configuration of one of the plugins
measureConfig.plugins["base_ClassificationExtendedness"].fluxRatio = 0.985

# Disable aperture correction, which requires having an ApCorrMap attached to
# the Exposure (it'll warn if it's not present and we don't explicitly disable it).
measureConfig.doApplyApCorr = "no"

# Actually create the Task.  This initializes all the plugins and that defines the
# rest of the schema
measureTask = lsst.meas.base.SingleFrameMeasurementTask(
    config=measureConfig,
    schema=schema
)
```

Running SingleFrameMeasurementTask

```
# Create a SourceTable from the Schema.  SourceTable is somewhat misleadingly named; it's
# really a factory for SourceRecords, not a container for them.
table = lsst.afw.table.SourceTable.make(schema)

# We pass the SourceTable and an Exposure to SourceDetectionTask.run(), and it'll return
# a SourceCatalog with empty records for the parents only.  Those will have Footprints
# attached to them.
detectResult = detectTask.run(table, exposure)
catalog = detectResult.sources

# We then pass the exposure and the catalog to the SourceDeblendTask, which adds new
# records for all the children, and attaches HeavyFootprints to them.
# Annoyingly, we have to pass the psf separately (DM-3987)
deblendTask.run(exposure, catalog, psf=exposure.getPsf())

# Finally, we can now run measurement.  Note the transposed argument order :/
measureTask.run(catalog, exposure)
```

Writing a `SingleFramePlugin` (in Python)

```
class BoxFluxConfig(lsst.meas.base.SingleFramePluginConfig):
    <new code here>

@lsst.meas.base.register("ext_BoxFlux")
class BoxFluxPlugin(lsst.meas.base.SingleFramePlugin):

    ConfigClass = BoxFluxConfig

    @classmethod
    def getExecutionOrder(cls):
        return cls.FLUX_ORDER

    def __init__(self, name, schema, metadata):
        lsst.meas.base.SingleFramePlugin.__init__(self, name, schema, metadata)
        <new code here>

    def measure(self, measRecord, exposure):
        <new code here>
```

Writing a `SingleFramePlugin` (in Python)

```
class BoxFluxConfig(lsst.meas.base.SingleFramePluginConfig):

    width = lsst.pex.config.Field(
        dtype=float, default=50,
        doc="approximate width of rectangular aperture"
    )

    height = lsst.pex.config.Field(
        dtype=float, default=50,
        doc="approximate height of rectangular aperture"
    )
```

Writing a `SingleFramePlugin` (in Python)

```
def measure(self, measRecord, exposure):

    centroid = <get a previously-measured centroid from measRecord>

    # Create a single-pixel box
    point = lsst.afw.geom.Point2I(centroid)
    box = lsst.afw.geom.Box2I(point, point)

    # Grow the box to the desired size
    box.grow(lsst.afw.geom.Extent2I(self.config.width//2, self.config.height//2))

    # Horrible syntax to create a subimage.  Can't use [] because it doesn't pay
    # attention to xy0 :-(
    subMaskedImage = exposure.getMaskedImage().Factory(
        exposure.getMaskedImage(),
        box,
        lsst.afw.image.PARENT
    )

    # compute the flux by extracting and summing NumPy arrays.
    flux = subMaskedImage.getImage().getArray().sum()
    fluxSigma = subMaskedImage.getVariance().getArray().sum()**0.5

    <stuff the results into measRecord>
```

Writing a `SingleFramePlugin` (in Python)

```
def __init__(self, config, name, schema, metadata):
    lsst.meas.base.SingleFramePlugin.__init__(self, config, name, schema, metadata)

    # Get a FunctorKey that can quickly look up the "blessed" centroid value.
    self.centroidKey = lsst.afw.table.Point2DKey(schema["slot_Centroid"])

    # Add some fields for our outputs, and save their Keys.
    doc = "flux in a {0.width} x {0.height} rectangle".format(self.config)
    self.fluxKey = schema.addField(
        schema.join(name, "flux"), type=float, units="dn", doc=doc
    )
    self.fluxSigmaKey = schema.addField(
        schema.join(name, "fluxSigma"), type=float, units="dn",
        doc="1-sigma uncertainty for BoxFlux"
    )
```

Writing a `SingleFramePlugin` (in Python)

```
def measure(self, measRecord, exposure):

    centroid = measRecord.get(self.centroidKey)

    # Create a sing                    x
    point
    box =

    # Grow the box to the desired size
    box.grow(lsst.afw.geom.Extent2I(self.config.width//2, self.config.height//2))

    # Horrible syntax to create a subimage.  Can't use [] because it doesn't pay
    # attention to xy0 :-(
    subMaskedImage = exposure.getMaskedImage().Factory(
        exposure.getMaskedImage(),
        box,
        lsst.afw.image.PARENT
    )

    # compute the flux by extracting and summing NumPy arrays.
    flux = subMaskedImage.getImage().getArray().sum()
    fluxSigma = subMaskedImage.getVariance().getArray().sum()**0.5

    measRecord[self.fluxKey] = flux
    measRecord[self.fluxSigmaKey] = fluxSigma
```

why not `measRecord[self.centroidKey]`?

Error Handling in Plugins

If we try running the plugin as it stands now, we get:

```
lsst::pex::exceptions::LengthError: 'Box2I(Point2I(-14,-15),Extent2I(51,51))
doesn't fit in image 81x81'
...
NotImplementedError: The algorithm 'BoxFluxPlugin' thinks it cannot fail, but it
did; please report this as a bug (the full traceback is above).
```

That's because we haven't overridden `fail()`, and the default implementation assumes the algorithm is infallible.

- If it's a misconfiguration or something else that will cause every measurement to fail on every single source, raise **`lsst.meas.base.FatalAlgorithmError`**.
- If it's a known failure mode, the plugin should set at least two flags: a general failure flag for the plugin, and a specific flag indicating what went wrong. That can be done in two ways:
  - Just set the flags in **`measure()`**.
  - Re-raise as **`lsst.meas.base.MeasurementError`**, and set flags in **`fail()`**.
- All other exceptions will trigger warnings, and **`fail()`** will be called to set the general failure flag.

Error Handling in Plugins

```
def __init__(self, config, name, schema, metadata):
    lsst.meas.base.SingleFramePlugin.__init__(self, config, name, schema, metadata)

    # Get a FunctorKey that can quickly look up the "blessed" centroid value.
    self.centroidKey = lsst.afw.table.Point2DKey(schema["slot_Centroid"])

    # Add some fields for our outputs, and save their Keys.
    doc = "flux in a {0.width} x {0.height} rectangle".format(self.config)
    self.fluxKey = schema.addField(
        schema.join(name, "flux"), type=float, units="dn", doc=doc
    )
    self.fluxSigmaKey = schema.addField(
        schema.join(name, "fluxSigma"), type=float, units="dn",
        doc="1-sigma uncertainty for BoxFlux"
    )
    self.flagKey = schema.addField(
        schema.join(name, "flag"), type="Flag",
        doc="general failure flag for BoxFlux"
    )
    self.edgeFlagKey = schema.addField(
        schema.join(name, "flag", "edge"), type="Flag",
        doc="flag set when rectangle used by BoxFlux doesn't fit in the image"
    )
```

Error Handling in Plugins

```
@lsst.meas.base.register("ext_BoxFlux")
class BoxFluxPlugin(lsst.meas.base.SinglePlugin):

    ConfigClass = BoxFluxConfig

    FAILURE_EDGE = 1

    @classmethod
    def getExecutionOrder(cls):
        return cls.FLUX_ORDER

    def __init__(self, confi
        ...

    def measure(self, measRec
        ...

    def fail(self, measRecord, error=None):
        measRecord.set(self.flagKey, True)
        if error is not None:
            assert error.getFlagBit() == self.FAILURE_EDGE
            measRecord.set(self.edgeFlagKey, True)
```

`error` is guaranteed to be either an instance of `MeasurementError` or None

Error Handling in Plugins

If we try running the plugin again, we get a warning:

```
measurement WARNING: Error in ext_BoxFlux.measure on record 1:
...
lsst::pex::exceptions::LengthError: 'Box2I(Point2I(-14,-15),Extent2I(51,51))
doesn't fit in image 81x81'
```

That's because we're still throwing a `LengthError` in `measure()`, and that means we're not setting our new edge flag.

Error Handling in Plugins

```
def measure(self, measRecord, exposure):

    ...

    # Horrible syntax to create a subimage.  Can't use [] because it doesn't pay
    # attention to xy0 :-(
    try:
        subMaskedImage = exposure.getMaskedImage().Factory(
            exposure.getMaskedImage(),
            box,
            lsst.afw.image.PARENT
        )
    except lsst.pex.exceptions.LengthError as err:
        raise lsst.meas.base.MeasurementError(err, self.FAILURE_EDGE)

    # compute the flux by extracting and summing NumPy arrays.
    flux = subMaskedImage.getImage().getArray().sum()
    fluxSigma = subMaskedImage.getVariance().getArray().sum()**0.5

    measRecord[self.fluxKey] = flux
    measRecord[self.fluxSigmaKey] = fluxSigma
```

Unit Transformations

- Measurement algorithms always produce outputs in raw units (pixels, dn).
- Each plugin also has an associated `TransformClass`, which implements the interface defined by `MeasurementTransform`. Most new plugins can either use one of the existing subclasses or a trivial subclass.
- `TransformTask` can be used to calibrate a catalog produced by `SingleFrameMeasurementTask`, by calling all of the `TransformClasses` associated with the plugins that were run.

Unit Transformations

```
class BoxFluxTransform(lsst.meas.base.FluxTransform):

    def __init__(self, config, name, mapper):
        lsst.meas.base.FluxTransform.__init__(self, name, mapper)
        mapper.addMapping(mapper.getInputSchema().find(name + "_flag_edge").key)


@lsst.meas.base.register("ext_BoxFlux")
class BoxFluxPlugin(lsst.meas.base.SingleFramePlugin):

    ...

    @classmethod
    def getTransformClass(cls):
        return BoxFluxTransform

    ...
```

Plugins can also be written in C++ (and in fact most of the current ones are). The steps are:

- Define a control struct (the C++ version of the config class).
- Inherit from `SingleFrameAlgorithm`, and override `measure()` and `fail()`.
- Swig both classes as usual.
- Call the `wrapSingleFrameAlgorithm` function at module scope (so it's executed when the module is imported).

Utility Classes for Plugins

We've been doing things the hard way. There are a number of utility classes in `lsst.meas.base` to make things easier:

- `FlagHandler` will manage a set of flag keys for different error conditions, and implement `fail()` for you. Unfortunately this can only be used from C++ at the moment (DM-4009).
- `SafeCentroidExtractor` and `SafeShapeExtractor` get centroid and shape values from previous measurements while handling flags appropriately. These depend on `FlagHandler`, though, so they can only be used from C++ as well.
- `FluxResult[Key]`, `CentroidResult[Key]`, and `ShapeResult[Key]` map simple structs to records and make it easier to create output fields for common types of measurements.

Forced Measurement

It's also important to be able to measure sources in one image while holding some properties (e.g. centroids) fixed to values measured in another image.

For this, there's `ForcedMeasurementTask` (and a `ForcedPlugin` base class for plugins).

These work pretty much the same way as their single-frame counterparts, with additional arguments to deal with the reference catalog.

`ProcessImageForcedTask` and its subclasses provide command-line drivers for forced measurement, and are probably easier to use than `ForcedMeasurementTask` itself when the reference catalog is an LSST stack output.

Records and Keys

The guts of a `BaseRecord` look something like this:

```
class BaseRecord {
    uint8_t * data;
    shared_ptr<Block> block;
    shared_ptr<BaseTable> table;
};
```

A record remembers the table that created it, and relies on it to know its own schema.

The guts of a `Key` look something like this:

```
template <typename T>
class Key<T> {
    size_t offset;
};
```

Fast, but potentially dangerous.
Not unusual in C++, scary in Python?

And record access is just:

```
template <typename T>
T & BaseRecord::operator[](Key<T> key) {
    return *reinterpret_cast<T*>(this->data + key.offset));
}
```

Records and Tables

The guts of a `BaseTable` look something like this:

```
class BaseTable {
    Schema schema;
    PropertyList metadata;
    shared_ptr<Block> block;
};
```

To create a new `Record`, we ask the block for the next available data pointer. If we're at the end of the block, we allocate a new one.

So a `Table` doesn't actually know about the `Records` it has already produced - it only knows about the `Schema`, some metadata, and the block of memory it will use to create future `Records`.

Records and Catalogs

A `BaseCatalog` is really just (with some syntactic sugar):

```
template <typename Record>
class Catalog {
    shared_ptr<Record::Table> table;
    vector<shared_ptr<Record>> vector;
};
typedef Catalog<BaseRecord> BaseCatalog;
```

- No guarantee the records are in the order they were allocated in.
- No guarantee that the records are from the same `Block`.
- *No guarantee that the records are even from the same `Table` or `Schema`!*

Not unusual in Python (trust the user not to mess it up), but a little scary in C++.

Record/Table Subclasses

- Usually, you'll work with `SourceRecord` instead of `BaseRecord`.
- Each `SourceRecord` has a `Footprint` (or `HeavyFootprint`).
- There's a "minimal schema" that all `SourceRecord` schemas have to start with.
- SourceRecord and SourceCatalog have special getters for certain predefined field names, called slots. e.g.:
  - `getX()` ➔ `get("slot_Centroid_x")`
  - `getPsfFlux()` ➔ `get("slot_PsfFlux_flux")`

There's also:

- `SimpleRecord` - the immediate base class of `SourceRecord`. Minimal schema includes a unique ID and a celestial coordinate. Used directly for reference catalog entries in astrometry.
- `ExposureRecord` - holds metadata about an `Exposure`, including its `Psf` and `Wcs`. Mostly used by `CoaddInputs` and `CoaddPsf`.
- `AmpInfoRecord` - used by the camera geometry module to store both structured and flexible information about amplifiers.
- `PeakRecord` - used to store peaks within a `Footprint`.

Columns from Catalogs

The single most important result of all of this complexity is the ability to do this:

```
array = catalog["field"]
```

That is, we can get a NumPy array view of a column.

This only works if the catalog is contiguous in memory.

If it isn't, just do:

```
catalog = catalog.copy(deep=True)
```

Catalog Indexing Magic

- Passing a string or a `Key` will extract a column (returning a `numpy.ndarray`).
- Passing an integer, a slice, or a boolean array as `x` in `catalog[x]` will index the rows (returning a `Record` or a subset `Catalog`).
- Unlike NumPy arrays:
  - You can't pass an array of row indices (it's just not implemented).
  - Boolean indices don't force a copy, so they generally return a noncontiguous `Catalog` (this is intentional - we want to make all copies explicit).

Adding/Removing Columns

- Once we've created a table, we can't change the `Schema`.
  - Whenever you ask a `Record`, `Table`, or `Catalog` for its `Schema`, you get a copy, so modifying it won't do what you want.
  - If you want to add or remove columns, you have to create a new `Catalog` with a modified `Schema` and copy data into it. That's what `SchemaMapper` is for.

Adding a Column with SchemaMapper

```
# Load the original catalog from disk
catI = lsst.afw.table.SourceCatalog.readFits("forcedI.fits")

# Create a SchemaMapper that maps from the given Schema to a new one
# it will create.  This does not set up any mappings.
mapper = lsst.afw.table.SchemaMapper(catI.schema)

# Add everything from the given Schema to output Schema, creating mappings
# as we go (that's what 'True') does.
# addMinimalSchema can only be called before creating any output fields;
# by putting the new fields at the beginning, we ensure Keys from the old
# Schema work with the new one.
mapper.addMinimalSchema(catI.schema, True)

# Add a new field to the output schema.
mapper.editOutputSchema().addField("new_field", type=int,
                                   doc="A new field added in a SchemaMapper")

# Create an empty catalog with the SchemaMapper's output schema.
newCat = lsst.afw.table.SourceCatalog(mapper.getOutputSchema())

# Add all records from the old catalog to the new one, using the SchemaMapper
# to copy values.
newCat.extend(catI, mapper=mapper)
```

Flag Fields

`Flag` (boolean) fields work a little differently than other fields, because they're packed into single bits.

```
template <>
class Key<Flag> {
    size_t offset;
    size_t bit;
};
```

```
bool BaseRecord::get(Key<Flag> key) {
    int64_t packed = *reinterpret_cast<int64_t*>(this->data + key.offset));
    return packed & (0x1 << key.bit);
}
```

- Flag column arrays are copies, not views.
- Only `get`/`set` are supported, not `[]`.

Slots and Aliases

We've already mentioned the "slot" system, which adds getters to `SourceRecord` and `SourceCatalog` for some predefined names. Those names are usually *aliases* to real field names.

A `Schema` holds its aliases in an attached `AliasMap`. Aliases are "just" string mappings, but they can do partial matching; any string that starts with the alias name will have its beginning replaced by the target of the alias. For instance, if we define an alias:

```
'slot_ApFlux' -> 'base_CircularApertureFlux_1_0'
```

we automatically get the following substitutions:

```
'slot_ApFlux_flux' -> 'base_CircularApertureFlux_1_0_flux'
'slot_ApFlux_fluxSigma' -> 'base_CircularApertureFlux_1_0_fluxSigma'
'slot_ApFlux_flag' -> 'base_CircularApertureFlux_1_0_flag'
```

FunctorKeys

`FunctorKeys` provide a mechanism to get first-class objects out of one or more fields in a `Record`.

They behave mostly like regular `Keys`, but have slightly syntax for:

- creating new fields
- getting a `FunctorKey` from a `Schema` by name

```
schema = lsst.afw.table.SourceTable.makeMinimalSchema()

# Add center_x and center_y fields, and return a FunctorKey
# that aggregates them.
pointKey = lsst.afw.table.Point2DKey.addFields(
    "center", "center position", "pixels"
)

# Retrieve a FunctorKey that aggregates the coord_ra and
# coord_dec fields that are part of the Source minimal schema.
coordKey = lsst.afw.table.CoordKey(schema["coord"])
```

- Once you have a FunctorKey, you can use it just like any other Key, with the following exceptions:
  - You can only use `get`/`set`, not `[]`
  - You can't get column arrays (what would that return?)
- You can also frequently get `Keys` to the individual fields a `FunctorKey` uses.

```
catalog = lsst.afw.table.SourceCatalog(schema)
record = catalog.addNew()

# Set both x and y from a Point2D objects.
record.set(pointKey, lsst.afw.geom.Point2D(5.0, 4.0))

# Get just the x coordinate
record.get(pointKey.getX())
```

A Summary of `afw.table` Idiosyncrasies

- `[record, table, catalog].schema` returns a copy... but that copy shares an `AliasMap` with the original (use `AliasMap.disconnectAliases()` to separate them).
- You can't use `[]` (only `get/set`) on `Flag` fields, and column arrays for `Flag` fields are copies, not views.
- Using an array of booleans to index a catalog returns a noncontiguous view; use `.copy(deep=True)` to get a contiguous copy that you can get column arrays from.
- A column array view of an `Angle` field returns a view that's radians with `dtype=float64`.

For More Information

- The afw.table reference documentation intro page.
- Try introspecting Python objects first - e.g. print a `Schema` to learn about its fields.
- Python docstrings and signatures are more likely to be useful than you're used to in the LSST stack, but they're still only present for some methods.
- The Doxygen reference documentation is quite complete, but often only applies to the C++ interface (because I've tried to customize the Python side).
- If something seems weird, don't hesitate to ask (probably at community.lsst.org).