講習会
「Python + Jupyter notebook による
光赤外天文データ解析入門」
テキスト

開催日：2019 年 8 月 29 日―30 日

主催：国立天文台天文データセンター

講師：中島 康 (一橋大学)

はじめての Jupyter notebook

はじめてのjupyter notebook

1. jupyter notebookを使ってみよう

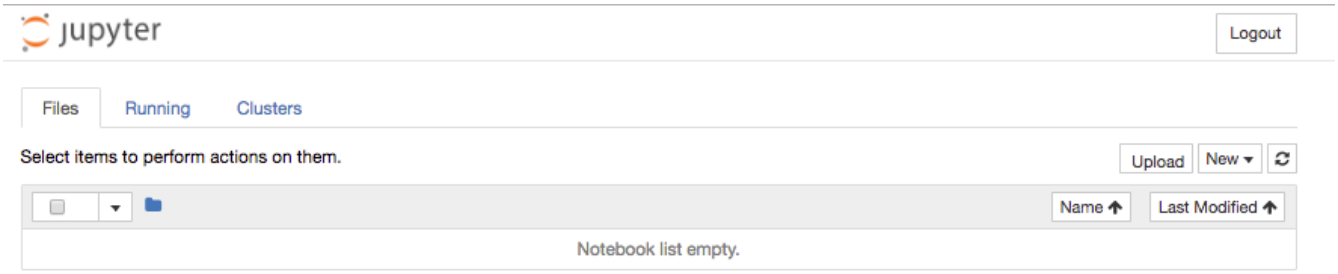
jupyter notebookを使うことで、対話的にデータ処理を行うことができます。 そして、その対話的な処理をメモと一緒にファイルとして保存することができます。

jupyter notebookは、もともとはipython notebookとよばれていたものをpython以外のプログラミング言語にも対応すべくグレードアップしたものです。julia, python, Rといった最近のデータサイエンスなんかで使われているプログラミング言語から文字をとってjupyterという名前が作られました。
https://github.com/jupyter/jupyter/wiki/Jupyter-kernels
によると40以上のプログラミング言語に対応しているとのことです。 もともとipython notebookだったので、jupyter notebookのファイルの拡張子は.ipynbになっています。

jupyter notebookの起動

コマンドラインで
jupyter notebook
を実行すると、デフォルトブラウザ内でjupyter notebookが起動します。

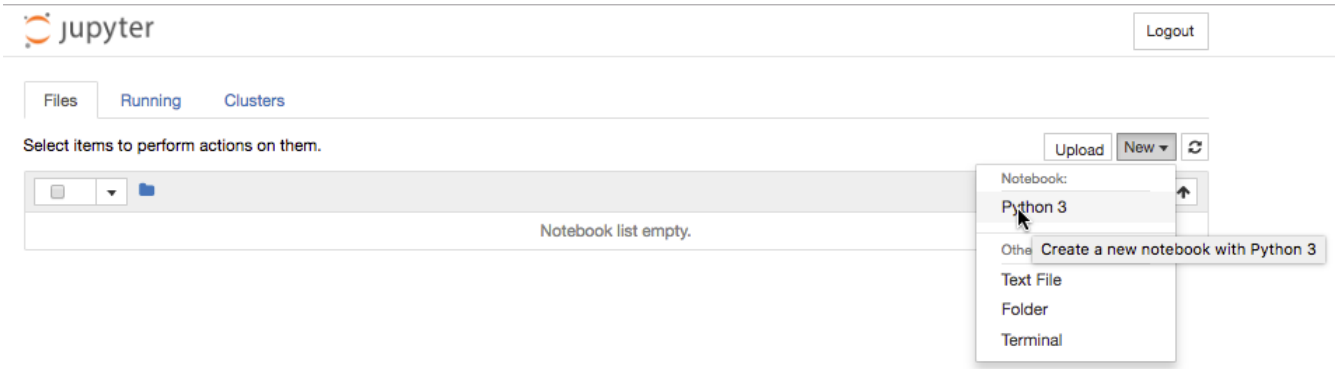
下のような画面がブラウザで表示されます。
(中身が空のディレクトリで起動すると、下のようにNotebook list emptyと表示されます。)



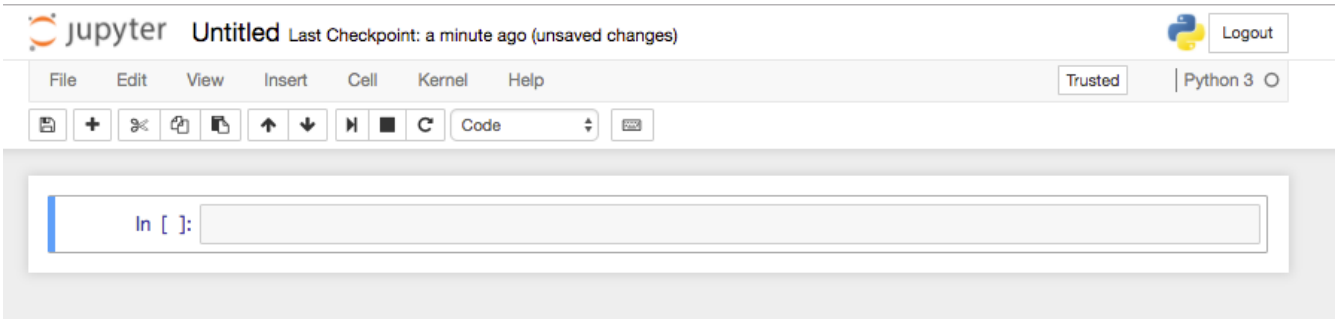
どこのディレクトリでも起動しますが、解析したいFITSファイルなどのデータがある作業ディレクトリで起動すると、データへのパスが簡単になります。 あるいは、私の場合は、自分の解析マシンにipynbファイルだけを集めたディレクトリを作っておき、 そこで起動してファイルを作成してから、作業ディレクトリに cd で移動することもあります。 この方法のよいところは、ipynbが自分の解析マシンの一箇所に集中して保存されるので、 ファイルの整理ができることです。

新規ノートブックの作成

右のNewからプルダウンメニューでPython3を選択します。

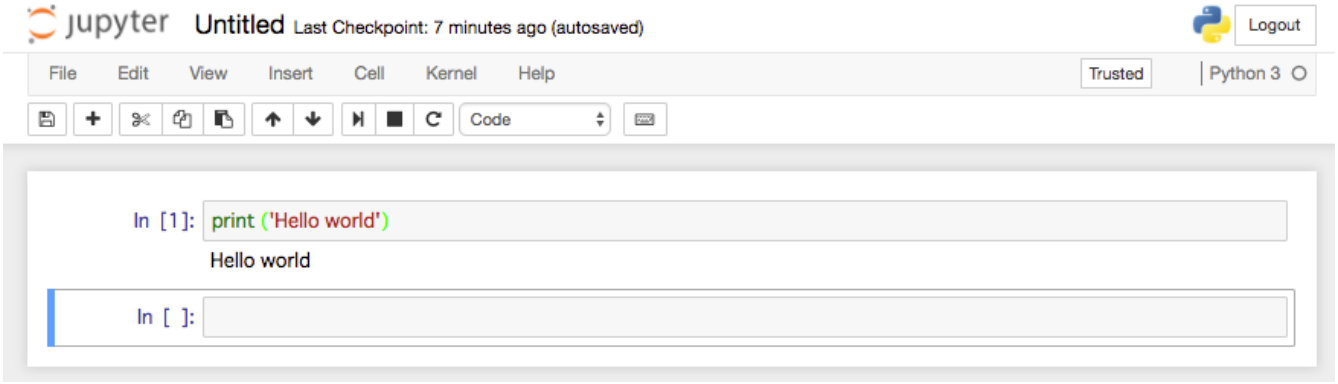


すると下のような新規ノートブック画面が出現します。



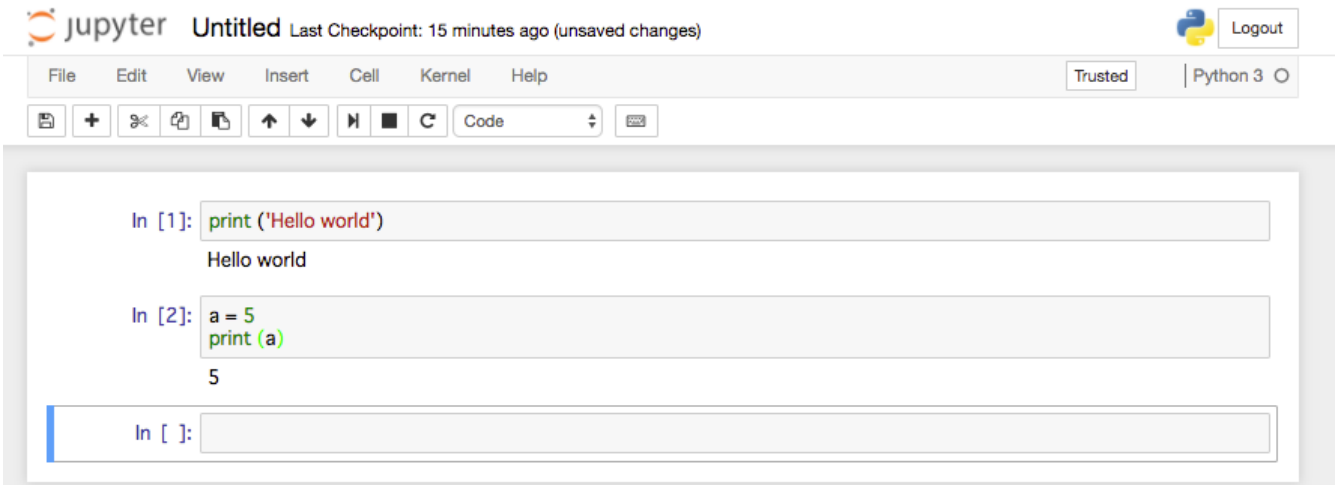
Pythonと対話してみましょう

Cellの中にPythonのコマンドを入力して**[shift]+[enter]**します。



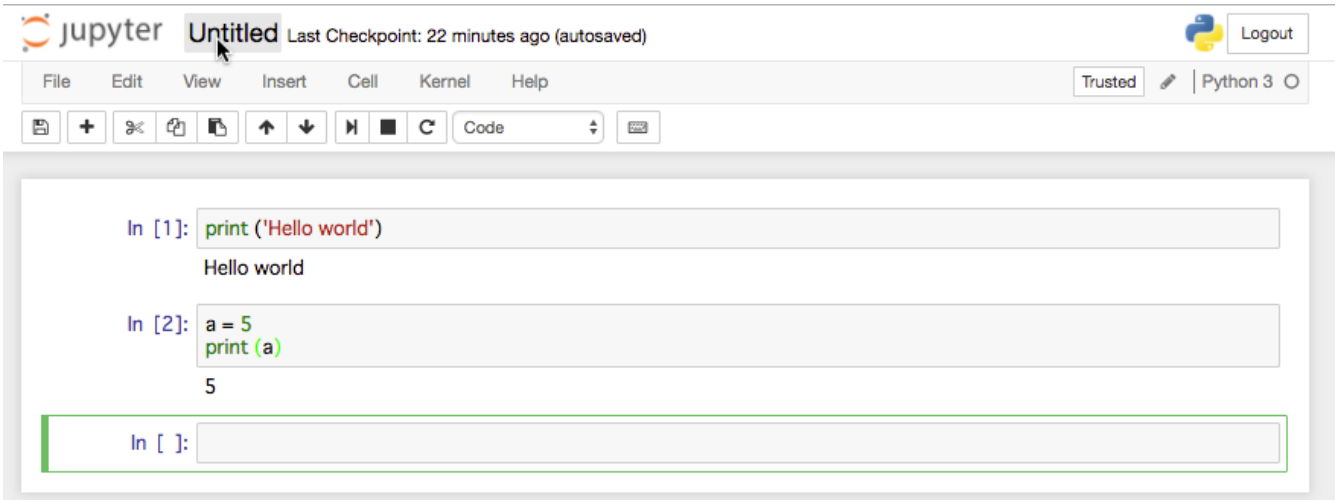
コマンドの入力に対する出力が、そのCellの下に表示されます。

コマンドが複数行にわたる場合、**[enter]**で改行します。 そのCellを実行する場合には、そのCellをマウスで選択した状態で**[shift]+[enter]**します。

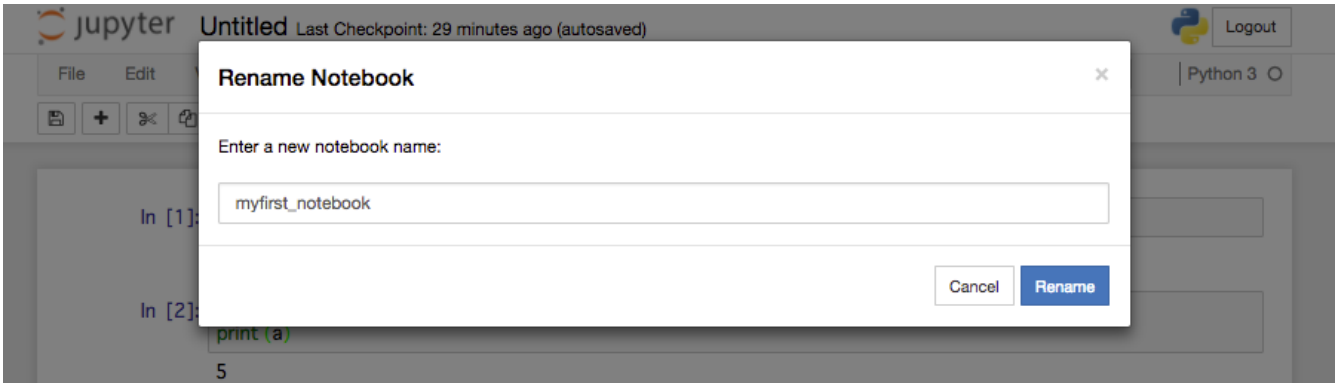


この対話を保存しましょう

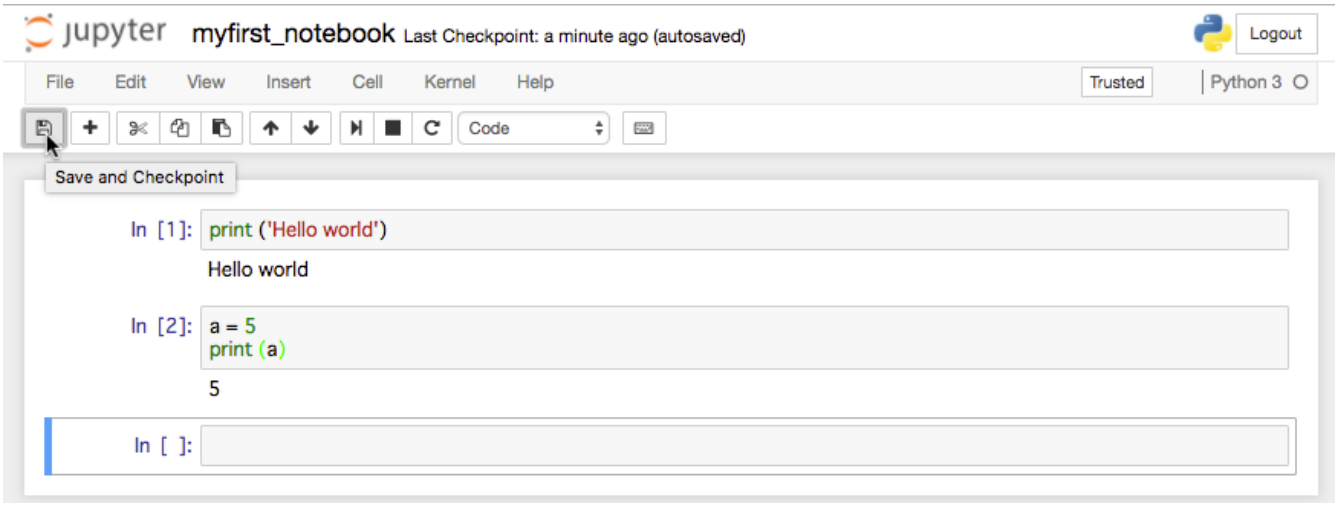
上部の**Untitled**のあたりにマウスカーソルを合わせると、背景がグレイにかわるのでクリックします。



ファイル名を入力して、Renameボタンを押します。(Jupyterのバージョンによっては、OKボタン)

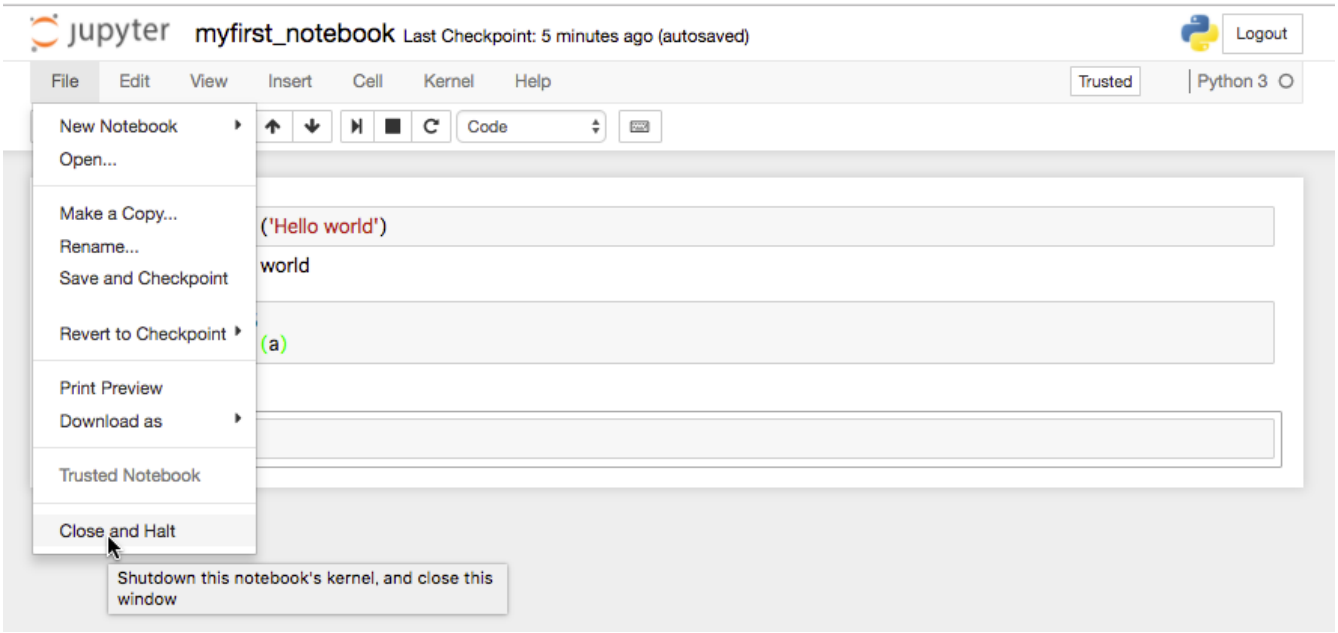


上部メニューバーの下のフロッピーのアイコンを押すと保存です。
デフォルトでは2分おきに自動保存されます。



ノートブックの終了

上部メニューバーの**File**のプルダウンメニューの一番下の**Close and Halt**を選んで終了。



jupyterを起動したときの画面に戻ります。ちゃんと名前をつけたファイルができていますね。

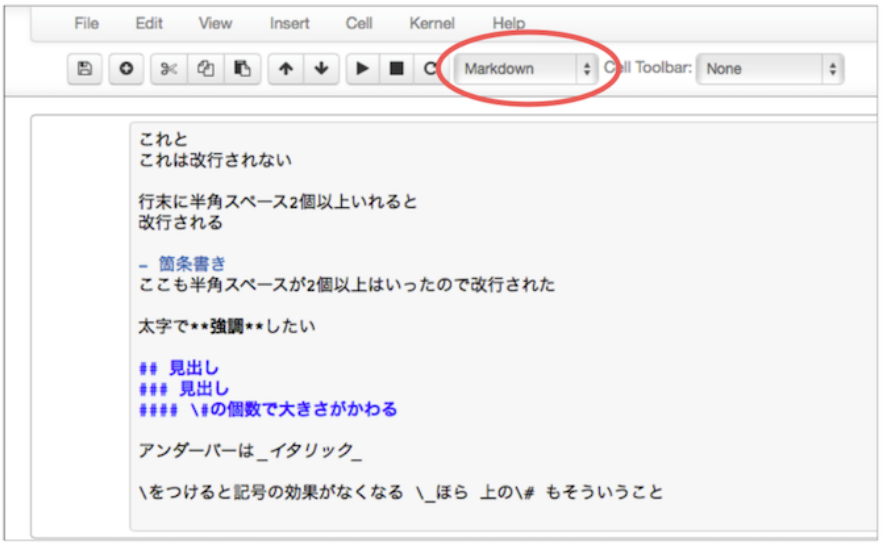
ここで、ファイル名のリンクをクリックすれば、そのノートブックの実行および編集が可能になります。

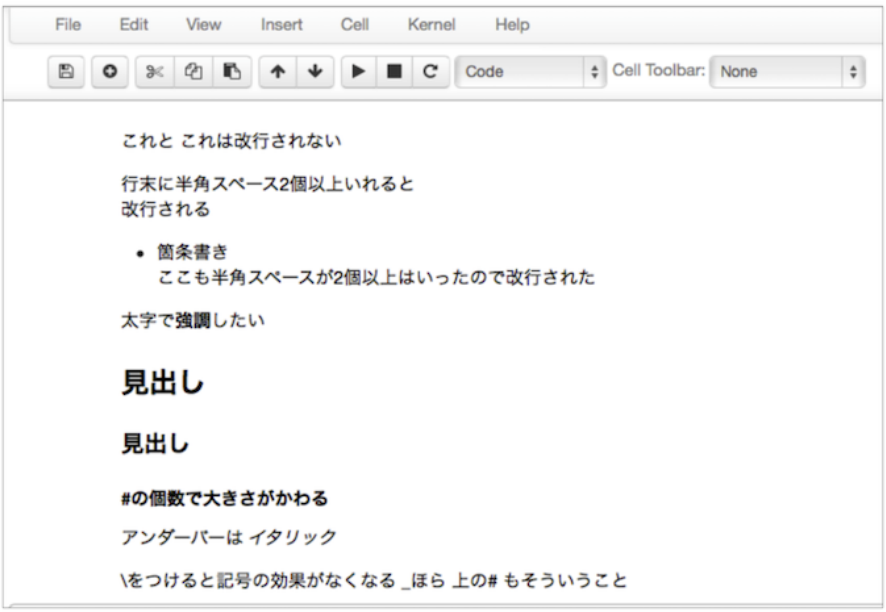
ブラウザを閉じて終了です。　ターミナルにて**Ctrl+C**でjupyter　notebookのプロセスを終了させます。ターミナルを閉じます。

2. メモを書き込む ~ markdown

マウスのカーソルでCellを選んだうえで、jupyter notebookの上部のデフォルトでは Code になっているところをMarkdownにすると、そのCellに文章を書き込むことができます。markdown形式で文章を書き込めます。markdownでは、マークアップ言語であるhtmlよりもかなり簡単な表記ルールで、文章に構造を持たせることができます。

文字列を書き込み、**[shift]+[enter]**するとmarkdownのルールに従ったレイアウトで表示されます。





「簡単な表記ルール」とはいえ、markdownは面倒だと思ったアナタへ。
シンプルテキストで書き込めばよいです。最低限のメモの記録はできます。

講習テキスト

講習０ ～ ７

講習0 --- Pythonとnotebook

この講習資料の使い方の説明を兼ねて、Pythonのインデントとモジュールの解説、 およびnotebookの変数読み込みと補完機能について解説します。

Pythonのインデントとモジュール

C言語などの他のプログラミング言語の経験があれば、Pythonの習得は簡単です。
ただし、いくつかPythonに特徴的な事項があります。

一番顕著なのは、インデント(字下げ)が意味を持つということでしょう。
forループの構造を紹介しながら説明します。

もう一つは、モジュールのimportとモジュール内の関数の使い方です。

インデント

forループ

C言語と比べるとずいぶん勝手がちがいます。

1. インデントされたブロックが、繰り返される部分です。
2. リストの各要素を順番に繰り返し変数に代入して使います。

In [1]:

```
for i in [0,1,2,3]:
    print (i)
```

```
0
1
2
3
```

In [2]:

```
total = 0
for i in range(6):  #   range()関数は0から引数の一つ前までの整数を順に返す
    print (i)
    total += i
print (total)
```

```
0
1
2
3
4
5
15
```

モジュールと関数

標準の組み込み関数(print()、range()、open()など)をのぞいて、関数を使用する際には、親となるモジュールをimportする必要があります。

mathモジュールのsqrt()関数を使う場合には次のようにします。

In [3]:

```
import math
```

In [4]:

```
math.sqrt(30)
```

Out[4]:

5.477225575051661

関数だけではなく、定数も含まれています。

In [5]:

```
math.pi
```

Out[5]:

3.141592653589793

In [6]:

```
a = math.e
```

In [7]:

```
print (a)
```

2.718281828459045

notebookの変数の読み込み

このipynbファイルを開いてすぐに、このすぐ上の **print(a)** のセル(In [7])を実行([Shift]+[enter])してみてください。 エラーがでます。 「aなんて定義されていない」、と言われちゃいます。 ファイルを開いただけでは、.ipynbの中の文字列が表示されただけなのです。 変数の読み込み、モジュールのimportなどはちゃんと実行してやらないといけません。

補完機能

jupyter notebookには補完機能があります。
すでに定義済みの変数、関数あるいはメソッドの最初の数文字をタイプして**[tab]**キーを押すと、候補が表示されます。ひとつしかない場合には、全て補完されます。 このnotebookの例だと、 tot までタイプして[tab]キーを押すと、totalまで補完されます。 math. までタイプして[tab]キーを押すと、mathモジュールで使える関数の候補が表示されます。

変数名には、(fooとかhogeではなく)その内容がわかるものをつけるのが望ましいですが、そうすると変数名が長くなりがちです。いちいちタイプするのが面倒になります。ですが、補完機能があればその煩わしさが軽減されます。結果として、より読みやすいコードがかけると思います。ぜひ、おためしあれ。

補足 : jupyter notebookからの返答とprint( )

jupyter notebookから何かを表示させるときに、二通りの方法があります。

1. オブジェクト(数値や文字列など)をセルに書き込んでそのまま **[shift] + [enter]** することで、jupyter notebookにその内容をこだまのように返させる方法。 (例 : In [5]のmath.pi)
2. **print( )** を使って、( )の中に与えたオブジェクトの内容を表示させる方法。(例 : In [7]のprint(a))

この二つの違いは以下です。

1. 前者はjupyter notebookの中でしか使えない「しくみ」であるが、後者はpythonのプログラムで一般的に使える。
2. 前者はオブジェクト型についての情報を与えてくれることがある。

「オブジェクト型についての情報を与えてくれることがある」とは下のような例です。

In [8]:

```
moji = '1'
```

In [9]:

```
moji
```

Out[9]:

'1'

In [10]:

```
print(moji)
```

1

ここで '1' は、' (シングルクォーテーション)で囲まれているので、文字列型のオブジェクトです。'1'という文字です。print( )の表示では文字型か数値型かの区別はつきません。

この資料では、簡易的に内容を確かめる時にはprint( )を使わず、プログラムの流れとして明示的にprintさせたいときにはprint( )を使う、といった使い方をします。

講習1 --- IRAFタスクをPythonから使ってみる

旧来のIRAFの操作は、ターミナルからCLコマンドラインを使って対話的に行うものです。
(IRAFでの使い方のおさらいは、このドキュメントの最後の付録1を参照)
この対話的な操作を、python + Jupyter notebookを使っても行うことができます。 ここでは、IRAFの基本的なタスク、**display, imexam, imstat, imarith** を使ってみます。

目標 : IRAFの基本タスクをjupyter notebookにて対話的に使えるようになる

- pythonからIRAFをモジュールとして呼び出す
- IRAFのタスクをpythonの関数として使う
- IRAFのタスクのパラメータを設定する
- IRAFのタスクから戻り値を取得する

注意 ここでは、IRAF Community Distribution の最新版 (IRAF 2.16.1+2018.11.01) の使用を前提とします。
NOAOで配布をしていたIRAFの最後のバージョンのものでは、多少挙動が異なることがあります。 例えば、ファイルへの上書きです。IRAF Community Distribution ではデフォルトで上書きを禁止していますが、 NOAO IRAFの最後のバージョンではデフォルトでは、上書きしようとしたデータが拡張形式として追加されるようになっています。

pyrafのための準備

pyrafは、pythonからIRAFのタスクを使うためのしくみです。
pyrafを(便利に)使うためには、

- ホームディレクトリにirafというディレクトリを作成
- そのディレクトリ内で mkirafを実行し、login.cl を作成
- 必要に応じてそのlogin.clを編集
  (今回のサンプルデータの場合サイズが大きいので、26行目あたりの'#set stdimage = imt800'を、例えば、'set stdimage = imt4096'にしておくとよいでしょう。冒頭の#を取り除き、800を4096に)

をしておきます。 こうしておくと、login.clでの設定がpyraf利用時に反映されます。また、~/iraf/uparm/に各タスク(imexam, imstatなど)のパラメータが保存されます。

補足 : ~/.iraf/login.cl がある場合には、そちらの設定が使用されます。
自分のホームディレクトリにIRAFをローカルインストールした場合、~/.iraf というディレクトリが作成されます。
~/iraf/login.clを編集したのに、それが反映されない場合は、 ~/.iraf/login.clを調べてみてください。

補足2 : stdimageの値を変更する場合、login.clを編集する以外の方法もあります。

```
from pyraf import iraf    #  iraf を import しておく
iraf.reset(stdimage='imt4096')
```

いつも同じサイズのFITSファイルを扱うのであればlogin.clに書き込んでおくのが楽です。iraf.reset()の方法も知っておくと、いろいろなサイズのFITSファイルを扱う場合に小回りがきいて便利です。

モジュールの読み込み

In [1]:

```
from pyraf import iraf
```

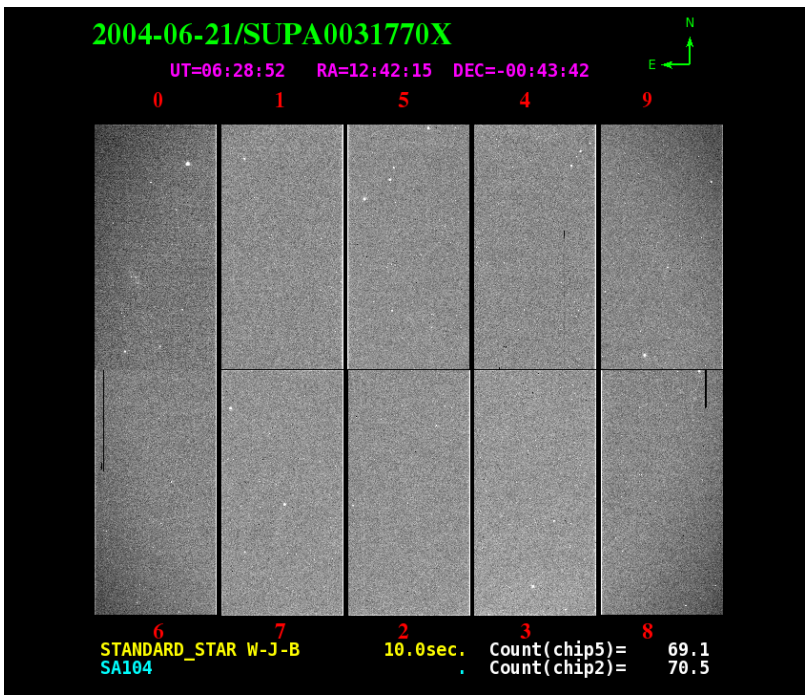
これでirafのタスクをpythonで関数として使うことができます。

サンプルデータ

サンプルデータとして、すばるSuprime Camで取得したデータを使います。 (SMOKAで公開されているデータです。) このファイルと同じディレクトリの data1 および data2 の中のFITSファイルです。(あらかじめ、uncompress.shを使ってbz2を解凍しておいてください)

この講習では、2004-06-21の観測データの一部を利用します。フィルターはBバンドで、ドームフラットおよび二種類のターゲットの生データです。Suprime Camは10枚のCCDから成ります。ファイル名(拡張子除く)の末尾の数字がCCDの番号を示しています。 data1には5番フレーム、data2には2番フレームのデータがあります。例えば、data1/SUPA00317705.fitsとdata2/SUPA00317702.fits は同じ積分のそれぞれ、5番フレームと2番フレームのデータです。講習のなかでは主に5番フレームを用いて説明をします。演習の中で2番フレームを使うことがあります。

FITSファイル	OBJECT	フィルター	積分時間(秒)
data1/SUPA003175[0-6]5.fits	ドームフラット	B	10
data1/SUPA00317705.fits	target1	B	10
data1/SUPA00317885.fits	target2	B	30
data2/SUPA003175[0-6]2.fits	ドームフラット	B	10
data2/SUPA00317702.fits	target1	B	10
data2/SUPA00317882.fits	target2	B	30



jupyterでのUnixコマンド

Unixコマンドを使い、カレントディレクトリおよびその中身を確認することができます。

In [2]:

```
pwd
```

Out[2]:

'/home/nakajima/adc2019python'

In [3]:

```
ls data1
```

```
SUPA00317505.fits  SUPA00317535.fits  SUPA00317565.fits
SUPA00317515.fits  SUPA00317545.fits  SUPA00317705.fits
SUPA00317525.fits  SUPA00317555.fits  SUPA00317885.fits
```

いくつかのUnixコマンドは、このように、jupyter notebookで直接使えます。使えないものもあります。その場合、! を冒頭につけて使用してやります。

In [4]:

```
! date
```

2019年 8月23日 金曜日 08時48分28秒 JST

それでは、IRAFのコマンド、display, imexam, imstatを使って、FITSファイルの表示やデータの吟味、統計量の測定をしてみましょう。

DS9にFITSデータを表示してみる iraf.display

別のターミナルから**ds9を立ち上げておきます。**
PyRAFでは、IRAFのタスクのdisplayは**irafモジュールのdisplay関数**として利用することができます。
iraf.display関数の引数としてファイル名(ファイルパス)とds9のフレーム番号を指定します。

では、target1の5番フレームをds9のFrame1に表示してみましょう。

In [5]:

```
iraf.display('data1/SUPA00317705.fits', 1)
```

z1=10012.29 z2=10105.81

ds9に、星がいくつか写っている視野が表示されましたね。
2番目の引数はなくても表示されます。ない場合には、デフォルトでFrame1に表示されます。

(**FAQ** ~/iraf/login.cl で set stdimage = imt800 のままだと視野の中心付近の800x800の領域しか表示されません。)

iraf.imexam

ローカルなピクセル値の統計や、星の特徴量などを調べるタスクです。
ds9にFITS画像を表示した状態で、次のコマンドを実行し、マウスカーソルを、星のないところにあてて **m** 、あるいは星にあてて **a** や **r** をタイプしてみましょう。終わるときには **q** をタイプしてください。

m : カウスカーソルを当てた部分の5x5ピクセルの範囲のカウント値の統計を表示します。バックグラウンドの値、ばらつきなどを調べるときに使います。
a : 星(点光源)の特徴量(ピーク値、fwhmなど)を表示します。
r : 星のradial profileを表示します。 星がサチっていれば一目でわかります。

In [6]:

```
iraf.imexam('data1/SUPA00317705.fits', 1)
```

```
#         SECTION    NPIX    MEAN   MEDIAN   STDDEV     MIN     MAX
[1114:1118,2031:2035]     25   10058.   10057.   8.062   10044.   10072.
 [797:801,2041:2045]     25   10059.   10056.   10.41   10047.   10092.
#   COL    LINE    COORDINATES
#    R    MAG    FLUX    SKY    PEAK    E   PA BETA ENCLOSED   MOFFAT DIRECT
1238.75 1913.35 1238.75 1913.35
 21.61  13.08  58384.  10059.   867.9 0.04   19 3.00    7.18    7.20   7.20
 779.62 2207.49  779.62 2207.49
 18.79  16.19   3330.  10059.   55.74 1.15  -49 4.73    6.57    7.23   6.56
```

imexamでは、**r** とか **e** コマンドを使うとさらに別のグラフィックウィンドウが現れます。ブラウザとかの後ろに隠れているかもしれません。

iraf.imstat

FITS画像の統計量を調べるタスクです。

In [7]:

```
iraf.imstat.unlearn()  # パラメータをデフォルト値に
```

iraf.タスク名.unlearn() を実行すると、そのタスクのパラメータをデフォルト値に戻してくれます。
irafでは、パラメータをカスタマイズするたびに、login.clと同じディレクトリにあるuparmというディレクトリの中のファイルに、パラメータが保存されます。 対話的に処理を行う際には、パラメータが保存されていると楽なこともあるのですが、 プログラミングによる処理を行う場合には、意図しないパラメータ値を知らずと使ってしまうこともありえます。
unlearn() しておき、パラメータを陽にプログラム内で指定すること　(後述の「変数としてパラメータ設定」参照)　をお勧めします。

補足 **:** 下のように　**iraf.unlearn('タスク名')**　でもOKです。

In [8]:

```
iraf.unlearn('imstat')
```

In [9]:

```
iraf.imstat('data1/SUPA00317705.fits')
```

```
#              IMAGE     NPIX     MEAN   STDDEV      MIN      MAX
 data1/SUPA00317705.fits   8528000   10059.    108.3    9911.    43932.
```

eparでパラメータ設定

iraf.epar('タスク名') で、パラメータ設定の画面が別ウィンドウで開きます。GUIでパラメータ設定ができます。
IRAFでもおなじみの方法ですね。

In [10]:

```
iraf.epar('imstat')
```

このように、GUIでもパラメータ設定ができるのですが、ここでは次の方法をおすすめします。

変数としてパラメータ設定

Cellでタスクの変数にパラメータを代入してやります。この方法だと、使ったパラメータがこのノートブックに残るので、あとになって「この処理でどんなパラメータ使ったっけ？」となったときに助けになります。

In [11]:

```
iraf.imstat.fields = 'midpt, mean, stddev'
iraf.imstat.lower = 0
iraf.imstat.upper = 20000
```

In [12]:

```
iraf.imstat('data1/SUPA00317705.fits')
```

```
#    MIDPT     MEAN    STDDEV
    10058.   10058.    41.62
```

どんな値がパラメータに入っているかを確認するには、パラメータ変数をprintしてやればよいです。

In [13]:

```
print (iraf.imstat.fields)
```

```
midpt, mean, stddev
```

関数の引数としてパラメータを指定する方法もあります。

In [14]:

```
iraf.imstat('data1/SUPA00317705.fits', fields='midpt, mean, stddev',
        lower=0, upper=20000)
```

```
#    MIDPT     MEAN    STDDEV
    10058.   10058.    41.62
```

結果の値を変数へ

上ではimstat()の結果が標準出力に表示されました。 それぞれの値を変数に保存するには次のようにします。

In [15]:

```
out = iraf.imstat('data1/SUPA00317705.fits', format='no', Stdout=1)
# format='no' でヘッダ行非表示、Stdout=1で戻り値を返す

print('outの中身 ', out) # outの中身。戻り値はリスト
v = out[0].split()  #  空白区切りの3つの数字をリスト化する
print('vの中身 ', v)  #  ここでは参考としてprintしておく

median = float(v[0])   # 文字列をfloatに変換しておく
mean = float(v[1])
stddev = float(v[2])

print (median)
print (mean)
print (stddev)
print (median + 3 * stddev) # floatに変換しておかないとここでおかしなことになる
```

```
outの中身  ['10058.19  10058.04  41.61855']
vの中身  ['10058.19', '10058.04', '41.61855']
10058.19
10058.04
41.61855
10183.04565
```

ここでのポイントは、iraf.imstatの引数パラメータの　**format**　と　**Stdout** の使い方です。
format='no'とすることで、 '# MIDPT MEAN STDDEV' のヘッダを表示しなくなります。
Stdout = 1 とすることで、結果を標準出力ではなく、リストを戻り値として返すので、変数に保存することができます。
戻り値のリストには、結果の各行が要素として保存されます。

ここでは、結果の出力は1行だけなので、out のリストには要素がひとつしかありません。
そこに空白区切で、結果の数値が文字列として並んでいます。 .split()で空白文字で分離して、変数 v にリストとして保存します。(上の結果の 「vの中身」 を参照)。 v[0], v[1], v[2]はこの段階では文字列なので、あとあとで計算に使いたい場合には、**float**関数を使って文字列を浮動小数点数に変換する必要があります。

iraf.imarith

ピクセル値の演算を行う関数です。

In [16]:

```
iraf.imarith('data1/SUPA00317705.fits', '-', 100, 'hoge.fits')
```

１番目の引数で指定したFITSの各pixelの値から100を引いて、hoge.fitsという名前のファイルとして保存します。

ここで **注意** することがひとつあります。

iraf.imarith の4番目の引数には演算結果を書き出すFITSファイル名を指定するのですが、そのファイル名として別の既存のファイル名を指定してしまうと、長いエラーメッセージの最後に、
ERROR (820, "Operation would overwrite existing image (hoge.fits) ")
のように怒られてしまい、その処理は実行されません。 別の既存のファイルに上書きできないのです。

ただし、１番目の引数のFITSファイル**自身**を上書き更新する場合は例外です。

In [17]:

```
iraf.imarith('hoge.fits', '+', 100, 'hoge.fits')
```

この上書きの振る舞いは、ここ5-6年くらいでコロコロと変わっています。NOAOのIRAFで一度変わって、IRAF Community Distributionでまた変わりました。（変わったというか元に戻った）　各自手持ちのPCに古いIRAFが入っている場合には挙動が講習会のときと変わるかもしれませんので注意してください。

helpドキュメント

このノートブック内でhelpを読むこともできます。
helpの表示が縦に長すぎる場合、左の余白部分(In[]: の下あたり)をクリックするとスクロールバーつきウインドウ表示になります。 (私の環境でブラウザChromeの場合にはデフォルトでスクロールバー表示されます。)

In [18]:

```
#iraf.help('imstat')  #  Gitlabでは表示が長くなってしまうのでコメントアウトしておきます
```

演習1

data1/SUPA00317885.fitsはtarget2の生データです。
新しいノートブックファイルを作成し、

1. imexamでバックグラウンドの値とばらつき、星の特徴量、を調べる。
2. imstatでカウント値のmedian, mean, standard deviationを求める。
3. imarithを使って、全pixelからmedian値を引く。

付録1. IRAFタスクの使い方 ~ 旧来のやり方

IRAFのコマンドラインでの各タスク、display, imexam, imstat, imarithの使い方をおさらいしておきます。

1. ターミナルを開きます。
2. ds9を起動します。(displayとimexamのために必要です)
   % ds9 &　　(%はコマンドプロンプト)
3. login.clのあるディレクトリで IRAFを起動します。
   *% cl *
4. display タスクで ds9のframe1にFITSを表示します。
   *cl> display example.fits 1 *
5. imexamタスクでバックグラウンドの値や、星の特徴量を調べます。
   (ds9にFITSが表示されていること)
   cl> imexam example.fits 1
   ds9上でカーソルが丸くなっています。
   - 星がない場所にカーソルをあてて、mのキーを押すと、カーソル位置を中心とする (デフォルトでは) 5x5 pixels の矩形領域のmedianやstandard deviationなどの 統計量を測ることができます。
   - 星にカーソルをあてて、aのキーを押すと、その星の特徴量(ピーク値やfwhm)を 測ることができます。
   - 星にカーソルをあてて、rのキーを押すと、星のradial profileを表示します。
6. imstatでFITSのpixel値の統計量を調べます。
   cl> imstat example.fits fields='midpt, mean, stddev'
   この例では、example.fitsのpixel値のメジアン、平均、標準偏差を求めます。

7. imarithでFITSの演算を行います。
   **cl> imarith example.fits - 1000 example2.fits **
   この例では、exmple.fitsの全pixel値から1000を引いて、example2.fits というファイルに書き出します。

IRAFのコマンドラインでは、上で見たように、display, imexamなどのタスク名のあとに、 空白を挟みながら、必要な引数をタイプして、実行します。

Pythonから使う場合には、IRAFのタスクは関数として使われます。
iraf.display('example.fits', 1) のように、( )の中に引数をタイプします。

講習2 --- IRAFで1次処理

IRAFのタスクを使って、**1 次処理**(生データからバイアスを引き、それをフラットで割る) を行いましょう。
ここでは、IRAFの基本タスク、**imstat**, **imarith**, **imcombine**を使います。

目標 : IRAFのタスク使って、jupyter notebookにてFITS画像の演算を行えるようになる。

- imstat, imarith, imcombineを連携させるひとまとまりのプログラムを作成する

ドームフラットでターゲットのフレームをフラット処理する (簡易版)

講習1で扱った'./data1/SUPA00317705.fits' はtarget1を観測した生データです。 これを、バイアス値を引いたあとに、**フラット**で割ることで、CCDの感度ムラと光学系の透過率のムラを補正します。 視野全体で一様な強度で光っている(と考えている)ものを観測してフラットを作成します。 データ処理の手引きには、ダークを引くことも書かれている場合がありますが、 (最近の) CCDではダークを生データから引くことはしません。 Suprime Camでも同様です。 (装置ごとにお作法が異なり得ますので、各自のデータ処理の際には、装置開発者などに確認してください。なお、地上観測の近赤外アレイでは、ダークを引いてフラットで割り、スカイバイアスを引くという処理が必要になります。)

フラットの作成

ドームフラットの生データからバイアス値を引いたのちに**規格化**(メジアン値で割ること)をしてフラットを作成します。通常は複数のフラットの平均から、より尤もらしくS/Nのよいフラットを作成します。

下では二段階に分けて練習をします。
まず、ドームフラット1枚だけからフラット作成します。**imstat**と**imarith**を使用します。
次に、ドームフラット7枚の平均から、より尤もらしくS/Nのよいフラット作成します。**imcombine**を使用します。

ドームフラット1枚だけを使う

'./data1/SUPA00317505.fits' はドームフラットのBバンドの生データです。 CCDのフレームには、観測した光に加えて、X方向に一様なバイアス値が加算されています。 この**簡易版**の処理では、そのバイアス値がY方向にも一様として処理をします。(付録 1 にY方向の依存も考慮に入れた手法を紹介します。)

ここではバイアス値を**オーバースキャン領域**から推定します。Suprime Camの5番フレームでは2049列目あたりから右側がオーバースキャン領域です。

まずはpyraf.irafをimportします。

In [1]:

```
from pyraf import iraf
```

IRAFのimstatを用います。

In [2]:

```
iraf.unlearn('imstat')
iraf.imstat.fields = 'midpt, mean, stddev'
iraf.imstat.nclip = 3
```

まず、オーバースキャン領域のメジアンを求めます。(不要ですが、mean, stddev(ばらつき)などの値も参考程度に見ておきます。)
FITSファイル名の後ろに、[2049:2080, *] をつけることで、Xが2049から2080、Yは全て(* はワイルドカード)、の矩形領域だけについて測定することができます。

In [3]:

```
iraf.imstat('./data1/SUPA00317505.fits[2049:2080, *]')
```

```
#    MIDPT     MEAN    STDDEV
     9984.     9984.     5.046
```

バイアス値は9984です。

次に、ドームフラットの光があたっている部分のメジアンを求めます。

In [4]:

```
iraf.imstat('./data1/SUPA00317505.fits[1:2048, *]')
```

```
#    MIDPT     MEAN    STDDEV
    18933.    18928.    210.1
```

ここでは、メジアン値として18933が求まりました。ここからバイアス値の9984を引いたものが、ドームフラット光のメジアン値です。

注意
Pythonに慣れていると、ここで

```
iraf.imstat('./data1/SUPA00317505.fits[:2048, *]')
```

のように、[ ]内の最初の1を抜かしてしまいがちです。 悪いことに、こうしてもIRAFはエラーや警告を出さないうえに、違う結果を出します。

次に**imarith**の出番です。バイアス値を引いてから、規格化してフラットを作成しましょう。

In [5]:

```
iraf.imarith('./data1/SUPA00317505.fits', '-' , 9984, 'bflatn5a.fits') # バイアス値を引く
iraf.imarith('bflatn5a.fits', '/' , 8949, 'bflatn5a.fits') # 規格化  18933 - 9984 = 8949、自身への上書き更新
```

このフラットでターゲットの生データを割ります。このときも、まず、生データからバイアス値を引きます。

In [6]:

```
iraf.imstat('./data1/SUPA00317705.fits[2049:2080, *]')
```

```
#    MIDPT     MEAN    STDDEV
     9989.    9989.    4.841
```

In [7]:

```
iraf.imarith('./data1/SUPA00317705.fits', '-', 9989, 'btarget1n5a.fits')
iraf.imarith('btarget1n5a.fits', '/', 'bflatn5a.fits', 'btarget1n5a.fits')
```

これでできました。 btarget1n5a.fitsをds9で表示して確かめてみましょう。

ドームフラット7枚を使う

iraf.imcombineの出番です。

'./data1/SUPA003175[0-6]5.fits' はBバンドのドームフラットです。

In [8]:

```
import glob # pythonの組み込みモジュール。ワイルドカードを使ったファイル処理など。
```

In [9]:

```
flist = glob.glob('./data1/SUPA003175[0-6]5.fits')
```

In [10]:

```
print(flist)
```

```
['./data1/SUPA00317515.fits', './data1/SUPA00317535.fits', './data1/SUPA003175
55.fits', './data1/SUPA00317525.fits', './data1/SUPA00317505.fits', './data1/SUPA0
0317545.fits', './data1/SUPA00317565.fits']
```

forループを使って、このリストからファイルを一つづつimstatに入力します。

In [11]:

```
for img in flist:
    iraf.imstat(img + '[2049:2080, *]')
```

```
#    MIDPT     MEAN    STDDEV
     9983.    9983.    5.039
#    MIDPT     MEAN    STDDEV
     9983.    9983.    5.032
#    MIDPT     MEAN    STDDEV
     9983.    9983.    5.056
#    MIDPT     MEAN    STDDEV
     9983.    9983.    5.036
#    MIDPT     MEAN    STDDEV
     9984.    9984.    5.046
#    MIDPT     MEAN    STDDEV
     9983.    9983.    5.038
#    MIDPT     MEAN    STDDEV
     9983.    9983.    5.071
```

各フレームにつき、2行の結果が表示されています。'# MIDPT MEAN STDDEV'のヘッダと、それぞれの測定結

果の数値です。

上では、glob.globで抽出したリストをいったん変数に代入しましたが、以下のように直接forループに入れても大丈夫です。

In [12]:

```
iraf.imstat.fields = 'midpt'  # どうせメジアンしか使わない

for img in glob.glob('./data1/SUPA003175[0-6]5.fits'):
   out1 = iraf.imstat(img + '[2049:2080, *]', format='no', Stdout=1)
   out2 = iraf.imstat(img + '[1:2048, *]', format='no', Stdout=1)
   print (out1, out2)
```

```
['9983.021'] ['18950.5']
['9983.005'] ['19029.06']
['9982.938'] ['18935.77']
['9982.991'] ['18963.1']
['9983.81'] ['18932.82']
['9982.896'] ['18995.43']
['9982.917'] ['18969.67']
```

out1, out2は、それぞれ、1つしか要素を持たないリストとして得られました。
下のように、リストの最初の要素を抽出することで値を得ることができます。

In [13]:

```
out1[0]
```

Out[13]:

'9982.917'

ただし、文字列ですので、このあとの計算で使うために、**float()**関数で数値(浮動小数点数)に変換してやります。

In [14]:

```
float(out1[0])
```

Out[14]:

9982.917

---

imcombineのおさらい

iraf.imcombine('im01.fits, im02.fits, im03.fits', 'out.fits', combine='median')
のように、コンバインするFITSファイル名をカンマ区切りで並べた文字列を一番目の引数として与え、出力するFITSファイル名を二番目の引数として与えます。平均のとりかた、madianかaverageかについてcombine=''として引数で与えます。

それでは、各ドームフラットからフラットを作成し、それらをメジアンでコンバインする一連のプログラムにしてみましょう。

In [16]:

```
iraf.imstat.fields = 'midpt'

num = 0
comstr = ''  # imcombineの一番目の引数となる文字列
for img in glob.glob('./data1/SUPA003175[0-6]5.fits'):

    out1 = iraf.imstat(img + '[2049:2080, *]', format='no', Stdout=1)  # オーバースキャン領域
    out2 = iraf.imstat(img + '[1:2048, *]', format='no', Stdout=1)  # 光のあたってる領域
    med1 = float(out1[0])  # 文字列を数値に変換
    med2 = float(out2[0])

    nflat =  'tmp' + str(num) + '.fits'   # それぞれのフラットを作成
    iraf.imarith(img, '-', med1, nflat)  # バイアス値をひく
    iraf.imarith(nflat, '/', med2 - med1, nflat) # バイアスを考慮して規格化

    num += 1
    comstr += nflat + ','  # imcombineの引数として与えるための文字列

print (comstr)  # なぜ下でcomstr[:-1]と、末尾の一文字を削除するか理解するためにあえて表示

iraf.imcombine(comstr[:-1], 'bflatn5.fits', combine='median')
iraf.imdelete(comstr[:-1])  # 中間ファイルを削除。お掃除お掃除。
```

```
tmp0.fits,tmp1.fits,tmp2.fits,tmp3.fits,tmp4.fits,tmp5.fits,tmp6.fits,

Aug 21 14:01: IMCOMBINE
  combine = median, scale = none, zero = none, weight = none
  blank = 0.
              Images
            tmp0.fits
            tmp1.fits
            tmp2.fits
            tmp3.fits
            tmp4.fits
            tmp5.fits
            tmp6.fits

  Output image = bflatn5.fits, ncombine = 7
```

comstr += nflat + ',' でどんどんファイル名を連結して作った文字列には、最後の , が余分なので、
comstr[:-1] として、末尾の一文字を削除したわけです。

これでドームフラットを7枚使ったフラットができました。
生データをこれで処理してやります。

In [17]:

```
iraf.imarith('./data1/SUPA00317705.fits', '-', 9989, 'btarget1n5.fits')
iraf.imarith('btarget1n5.fits', '/', 'bflatn5.fits', 'btarget1n5.fits')
```

補足

pythonには、os.remove というファイル削除の関数があります。

```
import os
os.remove('hoge.fits')
```

のように使います。ただしワイルドカードが使えません。

```
os.remove('tmp*.fits')
```

のようなことができないのです。　ですので、上では　**iraf.imdelete** を利用しました。

とはいえ、今後、脱IRAF化が進んでいくので他の方法も参考までに示しておきます。

(1) unixコマンドを使う

```
! rm tmp*.fits
```

一番楽だと思います。

(2) glob を使う。

```
for item in glob.glob('tmp*.fits'):
    os.remove(item)
```

演習2

2-1.
別ターゲットtarget2を観測した、'./data1/SUPA00317885.fits' について、バイアス引き+フラット割りの処理をしましょう。 これは5番フレームです。フィルターも同じBバンドなので、フラット割りには、'bflatn5.fits'が使えます。 (sampleディレクトリに用意してあります。)
この結果のフレームの名前を'btarget2n5.fits'とします。(後の演習で利用します)

2-2.
target1を観測した2番フレームの生データ'./data2/SUPA00317702.fits' について、 バイアス引き+フラット割りの処理をしましょう。 先ほどの5番フレームとは違い、これは2番フレームなので、2番フレームのためのフラットを作成する必要があります。
(1) './data2/SUPA00317502.fits' を規格化したものをフラットとして作成する。(1枚フラット)
(2) './data2/SUPA003175[0-6]2.fits' から平均のフラットを作成する。
(3) 上のどちらかのフラットを使って、バイアス引き後のフラット割りを行う。

注意 : 2番フレームはオーバースキャン領域が5番とは異なる。

付録1. バイアス値のY方向依存も考慮に入れる

今回使用するフラットでは0.1 x 数パーセントの違いしかありませんが、Y方向の依存も考慮に入れた方法を紹介しておきます。 ドームフラットのデータを1枚だけ使うケースを例にします。 irafのblkavgを使って、オーバースキャン領域の各lineの算術平均を求めます。(本当はメジアンがよいが。)

In [19]:

```
iraf.blkavg('./data1/SUPA00317505.fits[2049:2080, *]', 'bias1.fits', 32, 1)
```

bias1.fitsはサイズが(1,4100)の1次元データです。これをX方向に2080倍のばします。

In [20]:

```
iraf.blkrep('bias1.fits', 'bias2.fits', 2080)
```

このバイアスを生データから引いてやります。

In [21]:

```
iraf.imarith('./data1/SUPA00317505.fits', '-' , 'bias2.fits', 'bflatn5by.fits')
```

In [22]:

```
iraf.imstat('bflatn5by.fits[1:2048, *]', fields='midpt')
```

```
#    MIDPT
     8957.
```

In [26]:

```
iraf.imarith('bflatn5by.fits', '/', 8957, 'bflatn5by.fits')
```

バイアスのY方向依存も考慮に入れたフラットができました。
次にターゲットのフレームをフラットで割ります。
ここでもバイアス値のY方向依存を考慮にいれます。

In [24]:

```
iraf.blkavg('./data1/SUPA00317705.fits[2049:2080, *]', 'bias1t.fits', 32, 1)
iraf.blkrep('bias1t.fits', 'bias2t.fits', 2080)
iraf.imarith('./data1/SUPA00317705.fits', '-' , 'bias2t.fits', 'btarget1n5by.fits')
iraf.imarith('btarget1n5by.fits', '/', 'bflatn5by.fits', 'btarget1n5by.fits')
```

講習3 --- 星の測光

点光源の明るさを測定します。ここでは、IRAFのAPPHOTを用いて、アパーチャ測光を行います。

目標： IRAFのAPPHOTを使い、jupyter notebookにてアパーチャ測光ができるようになる。

- iraf.daofindとiraf.photを用いて、 星の検出と測光を行う
- iraf.txdumpを用いて、測光結果のリストを作成する
- 測光結果の較正を行う

準備のトリミング

講習2で作成したbtarget1n5.fitsをトリミングして、オーバースキャン領域など不要な部分を除いておきます。トリミングには、**iraf.imcopy**を使います。 (左端に明るい部分が見られます。これは何らかのバイアスがのっているものと思われるので、その部分(25列目まで)も除きます。)

In [1]:

```
from pyraf import iraf
```

iraf.imstatのときに出てきた矩形領域を指定する [x1:x2, *]をここでも使って、指定した矩形領域の部分だけ抜き出して、別のFITSファイルに保存します。

In [2]:

```
iraf.imcopy('btarget1n5.fits[25:2048, *]', 'btarget1n5trim.fits')
```

```
btarget1n5.fits[25:2048,*] -> btarget1n5trim.fits
```

パラメータ設定

iraf.apphot関連のタスクのためのパラメータを適切に設定する必要があります。

準備

- 星のサイズ(fwhm)を求めておく
- 背景のレベルとばらつきをもとめておく

ds9を立ち上げておき、iraf.displayとiraf.imexamで星のfwhmを調べておきます。

このあとの作業では、'btarget1n5trim.fits'に対して繰り返し処理をおこなうので、

In [3]:

```
targetfits = 'btarget1n5trim.fits'
```

とファイル名を変数に代入しておきます。

iraf.displayを実行する前に ds9を別ターミナルから起動しておいてください。

In [4]:

```
iraf.display(targetfits)
```

z1=28.35372 z2=115.3357

In [6]:

```
iraf.imexam(targetfits)
```

```
#   COL    LINE    COORDINATES
#     R    MAG    FLUX     SKY    PEAK    E   PA BETA ENCLOSED   MOFFAT DIRECT
 752.11 3381.95 752.11 3381.95
  21.59  10.96 412761.   70.92   5820. 0.05    5 10.0    7.13    7.42   7.20
 687.45 3180.93 687.45 3180.93
  22.07   9.34 1.845E6   77.92  25709. 0.05   16 13.2    7.18    7.52   7.35
1609.91 2403.08 1609.91 2403.08
  21.35  12.75  79099.    69.   1129. 0.05   27 4.96    7.01    7.57   7.12
 261.96 2855.91 261.96 2855.91
  22.58   8.99 2.541E6    79.4  34354. 0.06   18 3.80    7.32    7.68   7.53
 782.94 2801.10 782.94 2801.10
  21.65  13.65  34587.   69.07   492.7 0.09   57 6.46    7.06    7.44   7.22
```

丸に変身したカーソルを星にあてて、**a** を押してください。何個か測定したら **q** で終了です。

ENCLOSED, MOFFAT, DIRECT がfwhm(pixels)です。測り方がぞれそれ異なります。 ここでは、ざっくりとfwhm=7.0 pixel としておきます。

次に背景のmedianとノイズの評価をします。

In [7]:

```
iraf.imstat.unlearn()
iraf.imstat.fields = 'midpt, mean, stddev'
iraf.imstat.nclip = 3
```

iraf.imstat.nclip = 3として、3-sigma clipを3回イテレーションします。

In [8]:

```
iraf.imstat(targetfits)
```

```
#    MIDPT     MEAN    STDDEV
     68.84     69.1     7.368
```

メジアンは69でノイズは7.4とします。

パラメータの設定

apphotモジュールをimportします。

In [9]:

```
from iraf import digiphot
from iraf import apphot
```

補足
login.cl の記載内容によっては、

```
from iraf import digiphot
```

が不必要かもしれません。

次に、測光に必要な**パラメータを設定**します。
パラメータ設定の参考にした文献は、"A Reference Guide to the IRAF/DAOPHOT Package"です。
http://iraf.noao.edu/docs/photom.html (http://iraf.noao.edu/docs/photom.html)
からダウンロードできます。

In [10]:

```
# 上で測定した背景レベルとばらつき、fwhm
med = 69.
std = 7.4
fwhm = 7.0

iraf.apphot.unlearn()  # デフォルト値に戻しておく

iraf.apphot.datapars.datamax = 50000  # サチった星を数えない
iraf.apphot.datapars.readnoise = 10 # 検出器に特有な値
iraf.apphot.datapars.epadu = 2.5 # 検出器に特有な値 gain
iraf.apphot.datapars.itime = 10  # 積分時間

iraf.apphot.findpars.threshold = 7  # 7シグマ以上のものを検出せよ
iraf.apphot.findpars.sharphi = 0.8  # 星っぽくないものを除くため

# fwhmで決まるパラメータ
iraf.apphot.datapars.fwhmpsf = fwhm
iraf.apphot.centerpars.cbox = max(5.0, fwhm)
iraf.apphot.fitskypars.annulus = 3 * fwhm
iraf.apphot.photpars.apertures = 2 * fwhm

iraf.apphot.fitskypars.dannulus = 10.

# 背景のレベルとばらつきで決まるパラメータ
iraf.apphot.datapars.sigma = std
iraf.apphot.datapars.datamin = med - 5 * std

iraf.apphot.photpars.zmag = 27  # 等級のゼロ点

# IRAFと対話的(確認など)に行わないための設定
iraf.apphot.daofind.interac = 'no'
iraf.apphot.daofind.verify = 'no'
iraf.apphot.phot.interactive = 'no'
iraf.apphot.phot.verify = 'no'
iraf.apphot.phot.verbose = 'no'
```

まずは、**iraf.daofind**で星を検出させます。

In [11]:

```
iraf.daofind(targetfits, output='out1.coo')
```

どれが星として検出されたかFITS上にプロットしてみましょう。
まだ、ds9にtargetfitsがdisplayされたままであることを前提としています。

In [12]:

```
iraf.tvmark('1', 'out1.coo', mark='circle', radii='15, 16, 17', color=207)
```

radii='15,16,17'　としているのは線が細いからです。半径を少しずつ変えて3本引くと見える太さになりました。

In [13]:

```
#iraf.help('tvmark')
```

星じゃないものも検出されちゃってますが、ここでは気にせず、daofindの出力の'out.coo'を **iraf.phot**に読み込ませて測光します。

In [14]:

```
iraf.phot(targetfits, coords='out1.coo', output='out1.mag')
```

最初の引数は、測光する対象のFITSファイル名です。coords=で読み込ませる座標ファイルの名前、output=で結果を書き出すファイル名を指定します。

結果のファイルは、iraf.phot固有の形式で書き出されています。一つの星につき複数行の記載があり、このままでは少し扱いにくいです。

In [15]:

```
#cat 'out1.mag'  # gitlabでは長くなるのでコメントアウトしておきます
```

通常、必要なのは、xcenter, ycenter, mag, merr です。**iraf.txdump**を使って、それらだけを抜き出します。

In [16]:

```
iraf.txdump('out1.mag', fields='xc,yc,mag,merr')
```

```
1384.487  706.380  16.778  0.003
1394.061  921.854  17.593  0.005
724.860  986.570  15.692  0.001
412.063  1136.186  16.854  0.003
1256.533  1254.923  19.872  0.032
1614.107  1296.000  24.509  2.293
1.975  1563.123  INDEF  INDEF
325.259  1710.853  20.233  0.047
295.760  1891.867  INDEF  INDEF
1214.740  1913.393  17.615  0.005
164.208  2090.797  20.722  0.073
1519.869  2131.413  19.975  0.035
755.494  2207.461  20.578  0.061
1609.915  2403.143  17.285  0.004
888.608  2443.525  21.548  0.153
400.753  2577.342  INDEF  INDEF
2008.147  2638.538  16.281  0.002
782.873  2801.123  18.178  0.008
1781.279  2834.660  18.671  0.012
361.980  2855.986  13.514  0.000
```

INDEFなんてのもあります。これは、たまたまバッドピクセルが測光領域に含まれていた、視野の端である、あるいは、サチった星などです。 測光誤差を(例えば)0.2等以下のものだけに絞ることで、INDEFのものも削除

できます。 誤検出のものもここで削除できます。

iraf.txtdumpにて、expr='merr<=0.2'のように、引数としてexpr='条件式'を加えることで、リストの絞り込みができます。

In [17]:

```
iraf.txdump('out1.mag', fields='xc,yc,mag,merr', expr='merr<=0.2')
```

```
1384.487  706.380  16.778  0.003
1394.061  921.854  17.593  0.005
724.860  986.570  15.692  0.001
412.063  1136.186  16.854  0.003
1256.533  1254.923  19.872  0.032
325.259  1710.853  20.233  0.047
1214.740  1913.393  17.615  0.005
164.208  2090.797  20.722  0.073
1519.869  2131.413  19.975  0.035
755.494  2207.461  20.578  0.061
1609.915  2403.143  17.285  0.004
888.608  2443.525  21.548  0.153
2008.147  2638.538  16.281  0.002
782.873  2801.123  18.178  0.008
1781.279  2834.660  18.671  0.012
261.980  2855.986  13.514  0.000
1841.312  2952.747  20.019  0.040
687.466  3180.978  13.860  0.001
752.111  3381.985  15.486  0.001
```

さらに、引数としてStdout=''でファイル名を指定すると、その名前のファイルに書き出してくれます。

In [18]:

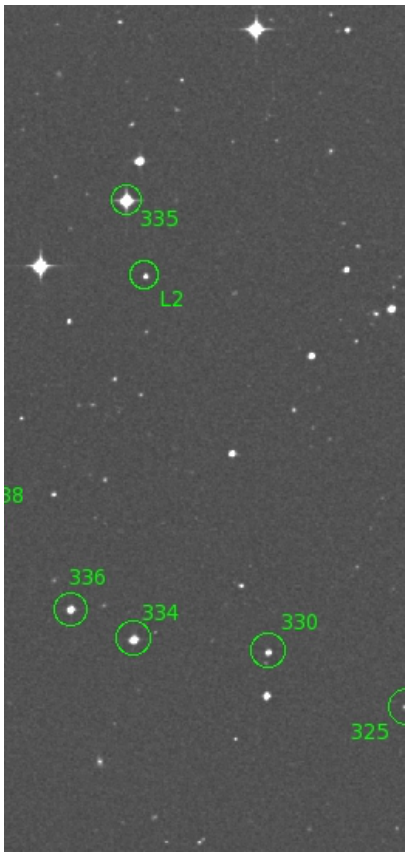
```
iraf.txdump('out1.mag', fields='xc,yc,mag,merr', expr='merr<=0.2', Stdout='result1.txt')
```

これで、星の(CCD上での)位置と機械等級がもとまりました。

測光値の較正

ここで求まったものは、機械等級です。等級のゼロ点を適当に設定したものです。 どうやって本当の等級に直せば良いでしょうか。

実は、この視野には標準星が写っています。Landolt(1992)のカタログの標準星のうち以下が写っています。



標準星	カタログB等級(エラー)	上記結果(機械等級)	3列 - 2列	カタログB-V	測定回数(カタログ)
SA104-330	15.894 (0.029)	17.593 (0.005)	1.699	0.594	15
SA104-334	13.998 (0.006)	15.692 (0.001)	1.694	0.518	24
SA104-335	12.292 (0.010)	13.860 (0.001)	1.568	0.622	4
SA104-336	15.230 (0.010)	16.854 (0.003)	1.624	0.830	14
SA104-L2	16.700 (0.033)	18.178 (0.008)	1.478	0.650	4

「3列 - 2列」の等級較正値にはばらつきがあります。 Landoltカタログで測定回数が10回以下のものは、ここでは、信頼性が低いとして採用しないことにします。 (それでもなお、SA104-336はSA104-330およびSA104-334と比べて0.07等も較正値が異なります。これはSA104-336のB-Vの値が他の2つと比べて大きいので、カラー変換の影響を考慮にいれなければいけないのかもしれません。)

結果として、較正値の平均は1.672等、標準偏差は0.03等となりました。 従って、result.txtの等級から1.672を引き、等級エラーには0.03等の誤差伝搬を加えておけばよいことになります。

In [19]:

```
import math  # 平方根の計算をするために math モジュールをimportする

with open('result1c.txt', 'w') as fout: # result1c.txtを書き出し先ファイルとする
    with open('result1.txt') as fin:  # result.txtを開く
        for line in fin:  # 1行づつ読み込み
            v = line.rstrip().split()  # rstrip()で改行コードを削除し、split()で空白文字で行を分割
            mag = float(v[2]) - 1.672 # 等級は3列目なので、それをfloatに変換して1.672を引く
            merr = math.sqrt(float(v[3])**2 + 0.03**2)
            print (v[0], v[1], mag, merr, file=fout) # 書き出し
```

補足

必須ではありませんが、フォーマット文を使い、

```
    print ('{} {} {:.3f} {:.3f}'.format(v[0], v[1], mag, merr), file=fout)
```

とするとmagとmerrが読みやすくなります。

次の講習でとりあげる**numpy**を使うと、上のテキストファイル(result.txt)の読み出し、計算、書き出しのプログラムが非常に簡単になります。

演習3

演習2-1で処理をした'btarget2n5.fits'で測光をしてみましょう。
このときも、オーバースキャン領域などの不要な部分を削除して行いましょう。

'btarget2n5.fits'の視野の中には測光標準星は写っていません。ただし、上のtarget1と近い時間に観測したデータですので、 等級ゼロ点は同じだと仮定し、上と同じ較正値(1.672)を使ってください。

講習4 --- Numpyの基本

この次の講習5では、IRAFを使わずにFITSファイルの処理をする方法を扱いますが、その準備としてNumpyの基本を知っておきましょう。

Numpyは、数値計算を効率よく処理するためのサードパーティモジュールです。 特に、多次元配列を取り扱う際に処理速度が速くなり、コードの表記も効率的になります。 pythonの標準の配列であるリスト型では処理が遅いため、科学計算ではNumpyの**ndarray**という多次元配列のデータ型を使います。Numpyはサードパーティモジュールですが、科学計算では標準的に使われます。このあと紹介するastropy.io.fitsおよびmatplotlibでも、このndarrayを採用しています。

以下、ndarrayの基本について説明し、numpyの基本的で'使える'関数について説明します。

目標 : ndarrayについて知り、その基本操作を身につける

- リスト型とndarrayの違いを理解する
- 1次元および2次元のndarray型データの作成方法を知る
- ndarray型データの基本操作を知る(ベクトル演算、スライシング)
- numpyの基本関数について知る

リスト型

まずはpythonの標準のリスト型を見てみましょう。

In [1]:

```
a = [1, 2, 3, 4]
b = [10, 20, 30, 40]
```

+ 演算子はリストとリストを結合します。* 演算子はリストの繰り返しを作成します。

In [2]:

```
a + b
```

Out[2]:

[1, 2, 3, 4, 10, 20, 30, 40]

In [3]:

```
a * 4
```

Out[3]:

[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]

ベクトル的に演算したい場合には、下のように、forループを回して各要素を取り出してから演算をする必要があります。

In [4]:

```
n = len(a)
c = [0] * n
for i in range(n):
    c[i] = a[i] + b[i]
print (c)
```

[11, 22, 33, 44]

Numpyのndarray

Numpyのndarrayではもっとすっきりした処理、**ベクトル処理**ができます。

ndarrayデータの作成 ~ 1次元

まずnumpyをimportします。

In [5]:

```
import numpy as np   # 一般的に np と省略される
```

numpy.array 関数を使って、pythonのリストをndarray に変換します。

In [6]:

```
an = np.array([1, 2, 3, 4])
bn = np.array([10, 20, 30 ,40])
```

ndarrayとリスト型の表示上の違いは下の通り。

In [7]:

```
an # print(an) ではなく、anとだけうつと、内容だけでなくobjectの型も含めた表記になる
```

Out[7]:

array([1, 2, 3, 4])

In [8]:

```
a # ちなみにリスト型だとこう
```

Out[8]:

[1, 2, 3, 4]

演算をしてみると、大きな違いがあります。
ndarrayどうしの + 演算はベクトル和の演算です。

In [9]:

```
cn = an + bn
cn
```

Out[9]:

```
array([11, 22, 33, 44])
```

掛け算についても、全ての要素に係数が掛かります。

In [10]:

```
an * 4
```

Out[10]:

```
array([ 4,  8, 12, 16])
```

ndarrayデータの作成 ~ 2次元

リストをカンマ区切で並べ、[ ] でくくります。 そのうえで、np.array()関数でndarrayにします。

In [11]:

```
myarr = np.array([[7, 3, 8], [13, 11, 16], [105, 121, 153]])
```

In [12]:

```
myarr
```

Out[12]:

```
array([[  7,   3,   8],
       [ 13,  11,  16],
       [105, 121, 153]])
```

二次元のインデックスを指定して、値を取り出してみます。 Pythonのインデックスは0から始まることに注意してください。

In [13]:

```
myarr[0, 1]
```

Out[13]:

```
3
```

In [14]:

```
myarr[1, 2]
```

Out[14]:

```
16
```

In [15]:

```
myarr[2, 0]
```

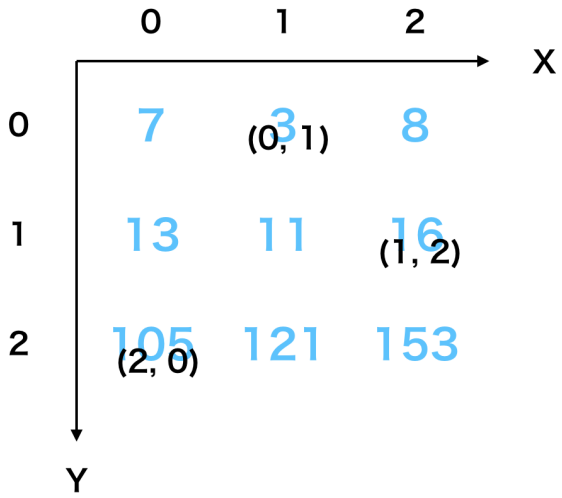
Out[15]:

105

myarrを表示すると、

```
array([[  7,   3,   8],
       [ 13,  11,  16],
       [105, 121, 153]])
```

のように現れますが、この横方向をX軸、縦方向をY軸(上下反転)とみなすと、 二次元インデックスの指定の仕方は、(x, y)ではなく、**(y, x)**のようになります。この点は、後のastropy.io.fitsのところでも再び触れます。



部分のとりだし

IRAFでは、FITSファイルの一部の矩形領域を取り出す時には、
[1: 2048, *] のようにして、「$1 \leq x \leq 2048$、 yは全て 」 の部分を取り出すことができました。

ndarrayの場合には異なります。
IRAFの [1: 2048, *] に相当する表記は [: , 0:2048] です。

Pythonのリストなどの配列において、 : を使った部分抽出のことを、スライシングといいます。
Pythonのスライシングでは、 [a : b] は a ≤ x < b を意味します。 (ここでa, bは整数。)
最後のbは範囲に含まれません。
そして、IRAFのインデックスは1からはじまりますが、Pythonでは0からはじまります。 ですので、0:2048 となります。

スライシングについて、例を見てみましょう。

In [16]:

```
myarr
```

Out[16]:

```
array([[  7,   3,   8],
       [ 13,  11,  16],
       [105, 121, 153]])
```

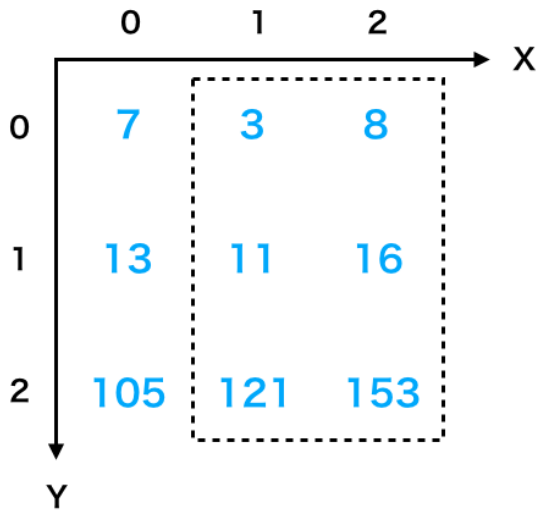
1番目のインデックスは全部で、2番目のインデックスは1以上のものを取り出す場合には、

In [17]:

```
myarr[:, 1:]   # インデックスの範囲が最後までの場合には、最後を省略できる。1:3 とせずに 1:  でOK。
```

Out[17]:

```
array([[  3,   8],
       [ 11,  16],
       [121, 153]])
```



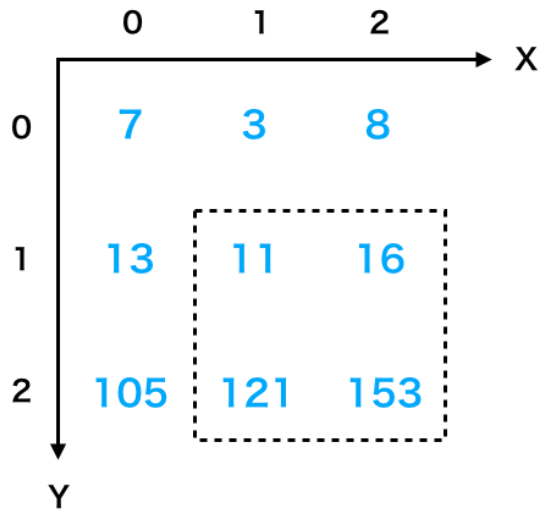
2番目のインデックスも1以上のものを取り出す場合にはさらに、

In [18]:

```
myarr[1:, 1:]
```

Out[18]:

```
array([[ 11,  16],
       [121, 153]])
```



次の講習でFITSの一部分を取り出す(トリミングする)場合にスライシングを使います。

とびとびの要素のとりだし

上記では、連続した部分のとりだしを見ました。 ndarrayではとびとびのインデックスの要素をとりだすことができます。

bnの中身は以下のとおりですが、

In [19]:

```
bn
```

Out[19]:

```
array([10, 20, 30, 40])
```

nparray形式でインデックスを指定すると、以下のようにとびとびの部分を取り出すことができます。

In [20]:

```
bn[np.array([0,2])]
```

Out[20]:

```
array([10, 30])
```

条件を満たす要素のとりだし、置き換え

ある条件を満たした要素だけを取り出すときに、以下のような書き方ができます。 非常に直感的です。

In [21]:

```
dn2 = np.array([-1, 20, 100, -10, -5, 80, -20, 100])
dn2[dn2 > 0]
```

Out[21]:

```
array([ 20, 100,  80, 100])
```

これを応用すると、ある条件を満たした要素だけ別の定数に置き換えることができます。

In [22]:

```
dn2 = np.array([-1, 20, 100, -10, -5, 80, -20, 100])
dn2[dn2 < 0] = -99
dn2
```

Out[22]:

```
array([-99,  20, 100, -99, -99,  80, -99, 100])
```

バッドピクセルマスク処理や外れ値の除外に使えます。

numpyの関数 (1) ~ 基本 + いくつか

numpyで用意されている関数は数多くあります。
cellの中で、np.とタイプしてタブキーを押すと候補が表示されます。あるいはnp.mでタブキーを押すとmで始まる関数の候補が表示されます。

In [23]:

```
np.max(cn), np.min(cn), np.mean(cn)
```

Out[23]:

```
(44, 11, 27.5)
```

numpy.where()

np.where( ) 関数は、( )内の条件を満たす要素のインデックスを返します。

In [24]:

```
dn3 = np.array([1, 1, 100, 1, 1, 100, 1, 100])
xx = np.where(dn3 > 2)
print(xx)
```

```
(array([2, 5, 7]),)
```

下の例では、2以上の値をとる要素に-99を代入します。

In [25]:

```
dn3[xx] = -99
```

In [26]:

```
dn3
```

Out[26]:

```
array([  1,   1, -99,   1,   1, -99,   1, -99])
```

これは、上で見た「ある条件を満たす要素の置き換え」と同じですね。

np.whereを使うと、その条件を満たす要素のインデックスを変数に保存することができます。プログラム中ではインデックスを変数に保存すると便利なこともあります。 次の講習で、3シグマクリップを使うときに登場します。

numpy.zeros( )

下のように、**numpy.zeros( )** を使って、全要素が0の5 x 5 の二次元配列を作成することができます。

In [27]:

```
data = np.zeros((5, 5))
```

In [28]:

```
data
```

Out[28]:

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [29]:

```
data[0, 1] = 2  #  [0, 1]の要素に2を代入する。
```

numpyに限らず、pythonではindexは0から始まります。C言語と同じです。
代入した結果を確かめて見ましょう。

In [30]:

```
data
```

Out[30]:

```
array([[0., 2., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

上でみたように、ndarrayでは (y, x)のように xとyのindexを逆にして要素が格納されます。

numpyの関数 (2) ~ 画像重ね合わせの準備

IRAFのimcombineを使わずに、astropy.io.fits + numpyで重ね合せる(つまり、複数画像の各ピクセルでのaverageやmedianを求めてcombineをする)ための準備として、numpy.stack(), numpy.average(), numpy.median()などを紹介します。

1次元

まずは1次元配列で見てみましょう。
3つの1次元配列、a, b, cがあるとき、

In [31]:

```
a1 = np.array([1, 2, 3])
b1 = np.array([2, 3, 4])
c1 = np.array([2, 2, 5])
```

numpy.stack()関数を使うことでこれらをまとめて一つ次元の高い配列にする(スタックする)ことができます。

In [32]:

```
s1 = np.stack((a1, b1, c1))
```

In [33]:

```
s1
```

Out[33]:

```
array([[1, 2, 3],
       [2, 3, 4],
       [2, 2, 5]])
```

そのうえで、配列を重ねてできた新しい次元の軸に沿ってmedianをとります。
上の表示例だと、縦方向にmedianをとります。
具体的には、**np.median([スタックされた配列], axis=0)** としてmedianの配列を得ます。

In [34]:

```
meddata = np.median(s1, axis=0)
```

In [35]:

```
meddata
```

Out[35]:

```
array([2., 2., 4.])
```

averageやsumも同様です。

In [36]:

```
np.average(s1, axis=0)
```

Out[36]:

```
array([1.66666667, 2.33333333, 4.        ])
```

In [37]:

```
np.sum(s1, axis=0)
```

Out[37]:

```
array([ 5,  7, 12])
```

axis=0を省略すると全ての要素について計算します。

In [38]:

```
np.average(s1)
```

Out[38]:

2.6666666666666665

2次元

2次元の場合も同様です。

In [39]:

```
a2 = np.array([[1, 2, 1],[20, 21, 13]])
b2 = np.array([[2, 2, 2],[22, 23, 15]])
```

In [40]:

```
print (a2)
print ()
print (b2)
print ()
print (np.average(np.stack((a2, b2)), axis=0))
```

```
[[ 1  2  1]
 [20 21 13]]

[[ 2  2  2]
 [22 23 15]]

[[ 1.5  2.   1.5]
 [21.  22.  14. ]]
```

(次に紹介する) astropy.io.fitsで読み込んだ画像の場合も同様にスタックをしてmedianやaverageをとります。

np.append()

最初からスタックする配列がそろっていれば上のように**np.stack()**を使うのが簡単ですが、通常のデータ処理の場合には、リストから次々にファイルを読みこんでスタックに追加していくことでより効率よくプログラミングができます。
次々にファイルを読み込んでスタックに追加する場合には、**np.append()**を使用します。

In [41]:

```
# a1 = np.array([1, 2, 3])
# b1 = np.array([2, 3, 4])
# c1 = np.array([2, 2, 5])

s2 = a1[np.newaxis, :]  #  s2 は array([[1, 2, 3]]) になる。np.newaxisは次元を増やすおまじない。
s2 = np.append(s2, b1[np.newaxis, :], axis=0)  #  s2は array([[1, 2, 3], [2, 3, 4]])
s2 = np.append(s2, c1[np.newaxis, :], axis=0)
```

In [42]:

```
s2
```

Out[42]:

```
array([[1, 2, 3],
       [2, 3, 4],
       [2, 2, 5]])
```

上で求めたs1と同じですね。

補足 np.empty()

プログラム中で、forループで回す場合には、まず空のndarrayを作っておき、そこに追加していくほうが都合のいいことが多いです。その場合には、**np.empty()**を使います。

In [43]:

```
s3 = np.empty((0, 2, 3))  # 0 x 2 x 3 の空の3次元ndarray
s3 = np.append(s3, a2[np.newaxis, :], axis=0)
s3 = np.append(s3, b2[np.newaxis, :], axis=0)
```

In [44]:

```
print (s3)
```

```
[[[ 1.  2.  1.]
  [20. 21. 13.]]

 [[ 2.  2.  2.]
  [22. 23. 15.]]]
```

In [45]:

```
s3.shape    # .shapeはndarrayの形状(各次元のサイズ)を取得するメソッド
```

Out[45]:

```
(2, 2, 3)
```

2 x 3　のndarrayが二つ重なったので、2 x 2 x 3のndarrayになりました。

In [46]:

```
print(np.average(s3, axis=0))
```

```
[[ 1.5  2.   1.5]
 [21.  22.  14. ]]
```

numpyの関数 (3) ~ テキストデータの読み込み

Numpyがなくても、Pythonの標準機能でテキストファイルの読み書きはできますが、数値テーブルであることが分かっていれば、**numpy.loadtxt()**を使ってもっと簡単に読み込み、ndarrayに保存することができます。

In [47]:

```
np.set_printoptions(precision=3, suppress=True)  # suppress=Trueで指数表示禁止。この行は必須ではない。
```

In [48]:

```
mlist = np.loadtxt('./sample/result1c.txt')
```

numpy.loadtxt()のオプションにて、ヘッダ行を無視、区切り文字(カンマ区切りやタブ区切り)、どの列を読み込むか、などを指定することもできます。

In [49]:

```
mlist
```

Out[49]:

```
array([[1384.487,  706.38 ,   15.106,    0.03 ],
       [1394.061,  921.854,   15.921,    0.03 ],
       [ 724.86 ,  986.57 ,   14.02 ,    0.03 ],
       [ 412.063, 1136.186,   15.182,    0.03 ],
       [1256.533, 1254.923,   18.2  ,    0.044],
       [ 325.259, 1710.853,   18.561,    0.056],
       [1214.74 , 1913.393,   15.943,    0.03 ],
       [ 164.208, 2090.797,   19.05 ,    0.079],
       [1519.869, 2131.413,   18.303,    0.046],
       [ 755.494, 2207.461,   18.906,    0.068],
       [1609.915, 2403.143,   15.613,    0.03 ],
       [ 888.608, 2443.525,   19.876,    0.156],
       [2008.147, 2638.538,   14.609,    0.03 ],
       [ 782.873, 2801.123,   16.506,    0.031],
       [1781.279, 2834.66 ,   16.999,    0.032],
       [ 261.98 , 2855.986,   11.842,    0.03 ],
       [1841.312, 2952.747,   18.347,    0.05 ],
       [ 687.466, 3180.978,   12.188,    0.03 ],
       [ 752.111, 3381.985,   13.814,    0.03 ],
       [1741.726, 4037.847,   19.611,    0.128],
       [1333.792, 4043.156,   11.89 ,    0.03 ],
       [1791.647, 4044.367,   16.449,    0.031],
       [ 650.189, 4085.125,   18.226,    0.048]])
```

ndarrayのデータをファイルに書き込むときは、**np.savetxt()**を使います。引数のfmtにてフォーマットを指定することもできます。

In [50]:

```
np.savetxt('result1calib.txt', mlist, fmt='%.3f')
```

演習4

(1) 下の二次元配列の演算をnumpyを用いて行ってください。

10	11	12
20	21	22
30	31	32

＋

100	110	120
200	210	220
300	310	320

(2) 下の3つの二次元配列のメジアンを、numpy.stack()とnumpy.median()を用いて求めてください。

10	11	12
20	21	22
30	31	32

100	110	120
200	210	220
300	310	320

20	22	24
40	42	44
60	62	64

付録1. ブロードキャスティング

ある2次元配列があるとき、そのどちらかの軸の大きさと同じ大きさの1次元配列の足し算、引き算をすると、もう一つの軸に沿って同じ演算をしてくれます。

In [51]:

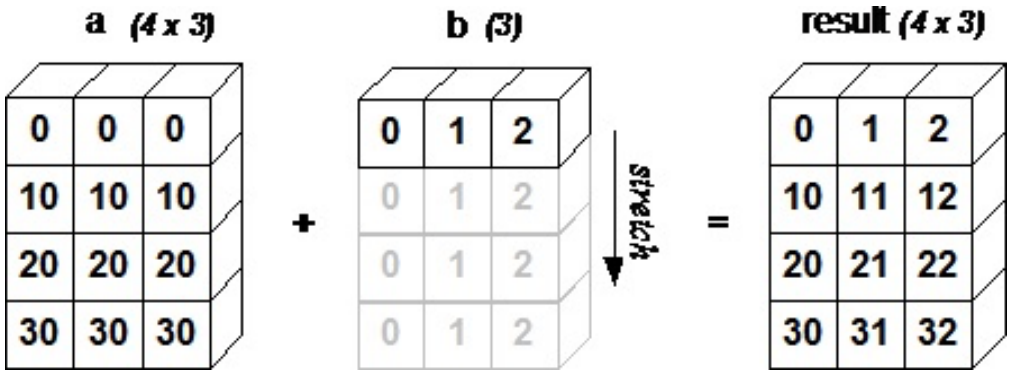
```
data1 = np.array([0, 1, 2])
data2 = np.array([[0, 0, 0], [10, 10, 10], [20, 20, 20], [30, 30, 30]])
```

In [52]:

```
data2 + data1
```

Out[52]:

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```



In [53]:

```
data3 = np.array([0, 100, 200])
data2 + data3
```

Out[53]:

```
array([[  0, 100, 200],
       [ 10, 110, 210],
       [ 20, 120, 220],
       [ 30, 130, 230]])
```

次のastropy.io.fitsのところで、Y方向に依存したバイアス値を引くケースで使います。

付録2. 配列について

Pythonの配列には、**リスト**と**タプル**があります。
リストは [] で囲まれ変更**可能**なオブジェクトです。
タプルは() で囲まれ変更**不可能**なオブジェクトです。

In [54]:

```
aa = [1, 2, 3, 4]
```

In [55]:

```
aa[2] = 10
```

In [56]:

```
aa
```

Out[56]:

```
[1, 2, 10, 4]
```

In [57]:

```
bb = (10, 20, 30, 40)
```

In [58]:

```
bb
```

Out[58]:

```
(10, 20, 30, 40)
```

In [59]:

```
bb[2] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-59-91c4ae2a694a> in <module>
----> 1 bb[2] = 100

TypeError: 'tuple' object does not support item assignment
```

タプルの要素を書き換えようとしたら怒られました。

講習5 --- astropy.io.fitsの基本

NOAOがIRAFの開発と管理から手を引き、この業界では脱IRAFの流れができています。
IRAFに依存しない天文解析用のモジュールが増えてきています。

astropy.io.fitsは、FITSデータの読み書きが可能なモジュールです。 元々、pyfitsと呼ばれていたものです。
astropyに吸収されました。
astropy.io.fitsの使い方は、
http://docs.astropy.org/en/stable/io/fits/index.html (http://docs.astropy.org/en/stable/io/fits/index.html)
が非常に参考になります。 例が多く記載されているのでわかりやすいです。

目標 : IRAFを使わずに、Pythonのモジュールのみで、FITSの読み書きと基本的な加工を行えるようになる。

- astropy.io.fitsを使ったFITSの読み書きの基本を理解
- numpyを使ったFITSデータの加工の基本を理解
- FITSのヘッダの加工の基本を理解

まずastropy.ioからfitsをimportしておきます。

In [1]:

```
from astropy.io import fits  # import astropy.io.fits as fits でも可
```

基本のgetdataとgetheader

まずは簡単なデータ読み出し方法。**fits.getdata( )**と **fits.getheader( )**です。 主にインタラクティブなケースで使われます。

まずはピクセル値をgetdataでndarrayとして読み出します。

In [2]:

```
data = fits.getdata('./data1/SUPA00317505.fits')  # ドームフラットの生データ
```

In [3]:

```
data
```

Out[3]:

```
array([[15798, 15697, 14527, ...,  9985,  9981,  9976],
       [18091, 33233, 30777, ...,  9991,  9987,  9978],
       [15287, 31092, 25091, ...,  9986,  9981,  9979],
       ...,
       [19053, 36375, 25385, ..., 10004, 10007,  9997],
       [16965, 34236, 24216, ...,  9998, 10001, 10000],
       [18178, 32768, 22645, ..., 10015, 10008, 10013]], dtype=uint16)
```

ndarrayとして読み出すので、numpyの関数が使えます。

In [4]:

```
import numpy as np
```

In [5]:

```
np.median(data), np.std(data), np.min(data), np.max(data)
```

Out[5]:

(18931.0, 1212.2635026788732, 9681, 38631)

オーバースキャンの部分も含めて統計をとっているので、標準偏差(np.std)が大きな値になっています。
オーバースキャンの部分を除いて統計をとってみましょう。

ndarrayなので、**[y, x]**の順に範囲を指定します。
[y_start:y_end, x_start:x_end] のように範囲をしていします。値を省略すると、「最初から」 あるいは 「最後まで」となります。左右両方を省略すると「全部」です。

In [6]:

```
subdata = data[:, :2045]
np.median(subdata), np.std(subdata), np.min(subdata), np.max(subdata)
```

Out[6]:

(18935.0, 409.56087590112406, 9739, 38594)

上限値と下限値を適切に設定し、3-sigmaクリッップすると標準偏差がもう少し小さくなります。

In [7]:

```
xx = np.where((subdata > 15000) & (subdata < 30000))
med = np.median(subdata[xx])
std = np.std(subdata[xx])

xx = np.where((subdata > med - 3 * std) & (subdata < med + 3 * std))
med = np.median(subdata[xx])
std = np.std(subdata[xx])
print ('{} {:.2f}'.format(med, std))   #   format()関数使ってみた
```

18934.0 213.65

次にgetheaderを使ってFITSヘッダを読み出します。

In [8]:

```
header = fits.getheader('./data1/SUPA00317505.fits')
```

ここで、header あるいは print(header) をcellで実行すると、FITSヘッダの中身が表示されます。

In [9]:

```
# header   # これをするとgitlabでは表示が長くなるのでここではコメントアウト
```

最初の10項目だけを表示させます。

In [10]:

```
header[:10]
```

Out[10]:

```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                 2080 / length of data axis 1
NAXIS2  =                 4100 / length of data axis 2
EXTEND  =                    F / FITS dataset may contain extensions
BZERO   =              32768.0 / offset data range to that of unsigned short
BSCALE  =                  1.0 / default scaling factor
BUNIT   = 'ADU     '           / Unit of original pixel value
BLANK   =               -32768 / Value used for NULL pixels
```

In [11]:

```
header['OBJECT']
```

Out[11]:

```
'DOMEFLAT'
```

このようにヘッダのキーワードを指定して、その値を取り出すことができます。

[小技]
実は、fits.getdata( )でヘッダも読み出すことができます。

In [12]:

```
tdata, thdr = fits.getdata('./data1/SUPA00317505.fits', header = True)
```

In [13]:

```
thdr[:10]
```

Out[13]:

```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                 2080 / length of data axis 1
NAXIS2  =                 4100 / length of data axis 2
EXTEND  =                    F / FITS dataset may contain extensions
BZERO   =              32768.0 / offset data range to that of unsigned short
BSCALE  =                  1.0 / default scaling factor
BUNIT   = 'ADU     '           / Unit of original pixel value
BLANK   =               -32768 / Value used for NULL pixels
```

データの加工と書き出し

このドームフラット1枚を使って、フラットを作成しましょう。 バイアス値がY方向に一様とする簡易版で行います。
ここでは、IRAFを使わずnumpyを使って行います。

In [14]:

```
omed = np.median(data[:, 2049:])  # オーバースキャン部分のメジアン
imed = np.median(data[:, :2048])  # 光が当たる部分のメジアン
data = ( data - omed ) / (imed - omed)  # ゲタを引き、規格化
```

整数データを割り算した結果、浮動小数点数になります。 デフォルトでは64ビットになります。

In [15]:

```
data.dtype
```

Out[15]:

```
dtype('float64')
```

無駄にファイルサイズが大きくなるので、32ビットに変更します。

In [16]:

```
data = data.astype(np.float32)
```

ヘッダのBITPIXは自動的に変更されます。

ヘッダのOBJECTの文字列を変更してみます。

In [17]:

```
header['OBJECT'] = 'B_FLAT'
```

ヘッダキーワードのBLANKはデータが整数のときのみ有効です。
それ以外のときにこの項目があると、アプリケーションによってはWarningがでます。
ここでは削除しておきましょう。

In [18]:

```
del header['BLANK']
```

このデータとヘッダを新しいFITSファイルに書き出します。**fits.writeto()**を使います。

In [19]:

```
fits.writeto('bflatn5pa.fits', data, header)  #  p for python
```

フラットなんて中間ファイルなので盛りだくさんなヘッダは不要ですよ、という時には、fits.writeto()の引数のheaderを省略すると、最低限必要なヘッダを勝手に作ってくれます。

In [20]:

```
fits.writeto('bflatn5pa_simple.fits', data)
```

元のFITSファイルに上書き更新する場合には**fits.update( )**を使います。

In [21]:

```
data = fits.getdata('bflatn5pa.fits')
data[:, 2049:] = -999999.  # オーバースキャン部の値を負の大きな値にしておく
fits.update('bflatn5pa.fits', data, header)
```

fits.getdata( ) や fits.getheader( )、fits.writeto( )などは簡単で便利 (これらはastropy.io.fitsの中で'convinience functions'と呼ばれている) なのですが、効率の悪いことをしています。その都度にファイルのオープンとクローズをしています。

その理由から、一般にプログラムコードの中でFITSファイルの読み書きをする場合には、**fits.open()** でファイルを開き、**.headerメソッド** と **.dataメソッド** を使ってヘッダーとデータを読み取ります。ここでは、fits.open()の講習は省略します。興味のある人は下の付録2を参照してください。

複数枚のドームフラットからフラットを作成

IRAFを使わずにcombineします。 numpy.median(), numpy.append()を使います。

In [22]:

```
import glob

stack = np.empty((0, 4100, 2080))  # 空の配列を作成

for img in glob.glob('./data1/SUPA003175[0-6]5.fits'):

    imdata = fits.getdata(img)
    omed = np.median(imdata[:, 2049:])
    imed = np.median(imdata[:, :2048])
    imdata = ( imdata - omed ) / (imed - omed)

    stack = np.append(stack, imdata[np.newaxis, :], axis=0)

immed = np.median(stack, axis=0)
immed = immed.astype(np.float32)

fits.writeto('bflatn5p.fits', immed)
```

ターゲットの生データのバイアス引きとフラット割り

target1の5番フレームの生データを、IRAFを使わずにastropy.io.fitsとnumpyで処理してみます。

In [23]:

```
imdata = fits.getdata('./data1/SUPA00317705.fits')
flat = fits.getdata('bflatn5p.fits')
flat[np.where(flat == 0.0)] = -9999  #  0での割り算を回避
omed = np.median(imdata[:, 2049:])
imdata = ( imdata - omed ) / flat
fits.writeto('btarget1n5p.fits', imdata)
```

トリミング

オーバースキャン領域はバイアス値の引き算が終わったあとは不要ですね。
切り取ってやりましょう。

In [24]:

```
imdata = fits.getdata('btarget1n5p.fits')
fits.writeto('btarget1n5ptrim.fits', imdata[:, :2048])
```

人工的な画像

ピクセル値が全部ゼロの10 x 10のFITSファイルを作り、いくつかのピクセルにだけ正の値を与えてやります。

In [25]:

```
imdata = np.zeros((10, 10), dtype='float32')  # 全て0の10x10のndarrayを作成
imdata[0, 1] = 100. #  (x,y)=(1,0)に100を代入。xとyが反転していることに注意
imdata[4, 6] = 50.

fits.writeto('my10x10.fits', imdata)
```

出来上がったFITSファイルをDS9で見てみましょう。そして、xとyが反転していることを確かめてください。

演習5

5-1. 上のmy10x10.fitsを作成し、DS9で見てxとyが反転していることを確かめてください。

5-2. astropy.io.fitsとnumpyを使って以下の処理をしてください。
(1) './data2/SUPA003175[0-6]2.fits'から2番フレーム用のフラットを作成。
(2) './data2/SUPA00317882.fits' について、バイアス引き+フラット割りの処理
(3) trimmingして(2)の結果からオーバスキャン部をとりのぞく。

付録1. バイアス値をY方向も考慮に入れる (numpy編)

numpyを使うと、IRAFでやるよりもすっきり書けます。
まずはフラットを作成します。

In [26]:

```
imdata = fits.getdata('./data1/SUPA00317505.fits')

bias = np.median(imdata[:, 2049:], axis=1) # X軸に沿ってmedianを計算。biasはサイズ4100の1次元配列
bias = bias.reshape(4100,1) # ブロードキャスティングできるように整形

imdata = imdata - bias   # ブロードキャスティングで引き算

med = np.median(imdata[:, :2045])

imdata = imdata / med
imdata = imdata.astype(np.float32)

fits.writeto('bflatn5pby.fits', imdata) # ヘッダは生データのものを継承しない
```

次に、生データもバイアス値のY方向依存を考慮して引きます。

In [27]:

```
flat = fits.getdata('bflatn5pby.fits')
flat[np.where(flat==0)] = -9999
tdata = fits.getdata('./data1/SUPA00317705.fits')
bias = np.median(tdata[:, 2049:], axis=1)
bias = bias.reshape(4100,1)

tdata = tdata - bias
tdata = tdata / flat

fits.writeto('btarget1n5pby.fits', tdata)
```

付録2. fits.open() を使ったファイルの読み書き

一般に、プログラムのコードの中でFITSファイルの読み書きをする場合には次のように、 fits.open() でファイルを開き、**.headerメソッド**と**.dataメソッド**を使ってヘッダとデータを読み取ります。

In [28]:

```
hdulist= fits.open('./data1/SUPA00317505.fits')
```

このFITSファイルがどのような構造になっているかを、**.info()**メソッドを使って調べることができます。

In [29]:

```
hdulist.info()
```

```
Filename: ./data1/SUPA00317505.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     150   (2080, 4100)   int16 (rescales to uint16)
```

一つのヘッダと画像データのセットだけが含まれています。
この場合、hdulist[0]のみにデータが含まれており、次のようにヘッダと画像データを読み取ります。
拡張FITSファイルの場合、hdulistの要素が複数あり、hdulist[1]などと指定します。

In [30]:

```
hdr = hdulist[0].header  # ヘッダを読み取り
imdata = hdulist[0].data  # データを読み取り
```

ヘッダの中身を10行目まで表示させます。

In [31]:

```
hdr[:10]
```

Out[31]:

```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                 2080 / length of data axis 1
NAXIS2  =                 4100 / length of data axis 2
EXTEND  =                    F / FITS dataset may contain extensions
BZERO   =              32768.0 / offset data range to that of unsigned short
BSCALE  =                  1.0 / default scaling factor
BUNIT   = 'ADU     '           / Unit of original pixel value
BLANK   =               -32768 / Value used for NULL pixels
```

In [32]:

```
omed = np.median(imdata[:, 2049:])  # オーバースキャン部分のメジアン
imed = np.median(imdata[:, :2048])  # 光が当たる部分のメジアン
imdata = ( imdata - omed ) / (imed - omed)  # ゲタを引き、規格化
hdulist[0].data = imdata.astype(np.float32)  # 32ビットに変更
```

これら処理をする過程で、ヘッダからBLANKというキーワードの行は消えます。

In [33]:

```
hdr[:10]
```

Out[33]:

```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                  -32 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                 2080 / length of data axis 1
NAXIS2  =                 4100 / length of data axis 2
EXTEND  =                    F / FITS dataset may contain extensions
BUNIT   = 'ADU     '           / Unit of original pixel value
DATE-OBS= '2004-06-21'         / Observation start date  (yyyy-mm-dd)
UT      = '05:20:20.053'       / HH:MM:SS.S typical UTC at the exposure (middle
UT-STR  = '05:20:20.053'       / HH:MM:SS.S UTC at the start exposure time
```

In [34]:

```
hdr['OBJECT'] = 'B_FLAT'
```

新しいファイルに書き出す場合には、**.writetoメソッド** を使います。

In [35]:

```
hdulist.writeto('bflatn5pb.fits')
```

```
WARNING: VerifyWarning: Invalid 'BLANK' keyword in header.  The 'BLANK' keyword is on
ly applicable to integer data, and will be ignored in this HDU. [astropy.io.fits.hdu.image]
```

ヘッダには'BLANK'はないのに、上のようなWarningが出てしまいます。 無視しましょう。

開いたファイルは最後には閉じておきましょう。

In [36]:

```
hdulist.close()
```

読み込んだファイルに上書き更新する場合には、 下のように、openするときにmode='update'を指定しておきます。変更がclose()のときにファイルの中身に書き込まれます。

In [37]:

```
f = fits.open('bflatn5pb.fits', mode='update')
data = f[0].data
data[:, 2049:] = -999999.  # オーバースキャン部の値を負の大きな値にしておく
f.close()
```

講習6 matplotlibの基本

matplotlibはデータ可視化のためのパッケージです。 http://matplotlib.org (http://matplotlib.org) のページのexamples のページ https://matplotlib.org/gallery/index.html (https://matplotlib.org/gallery/index.html) には膨大な数のサンプルがあり、どんなグラフを作成できるのかを見る事ができます。一度ご覧になるのをお勧めします。さらに、それぞれのサンプルコードも見る事ができます。 あるいは、基本的なものをまとめたtutorialsのページ https://matplotlib.org/tutorials/index.html (https://matplotlib.org/tutorials/index.html) も参考になります。

ここでは、光赤外撮像データ解析によく使いそうな、ヒストグラム、等級-エラープロット、FITSデータの表示の例を紹介します。

目標 : jupyter notebookでのmatplotlibによる可視化の基本を理解する

- ヒストグラムの作成
- データプロットの作成
- FITSの表示

inline表示

inline表示にしてやると、notebook内にグラフを表示することができます。 下のように宣言しておきます。

In [1]:

```
%matplotlib inline
```

matplotlibパッケージの中で最もよく使うモジュールはpyplotです。matplotlib.pyplot as plt の省略がよく使われます。

In [2]:

```
import matplotlib.pyplot as plt
```

測光結果データをグラフ化する

講習3で得られた測光結果、result1c.txt には、(列1) x座標、(列2)y座標、(列3) 等級、(列4)等級エラーが記されています。 numpy.loadtxt()を用いてこのデータを読み込みます。

In [3]:

```
import numpy as np
```

In [4]:

```
mlist = np.loadtxt('sample/result1c.txt')
```

ndarray形式で読み込まれます。array([[x, y, 等級, 等級エラー], [x, y, 等級, 等級エラー], ...])
ここでまずは欲しいのが、各星の等級と等級エラーのペアです。
このあとグラフ作成時の分かりやすさのために、magとmerrを分けて別々の配列にしておきます。

下のようにいくつか例を書き出してみると分かると思いますが、2番目のインデックスが2のものが等級で、2番目のインデックスが3のものが等級エラーです。

In [5]:

```
print (mlist[0,0], mlist[0,1], mlist[0,2], mlist[0,3])
print (mlist[1,0], mlist[1,1], mlist[1,2], mlist[1,3])
print (mlist[2,0], mlist[2,1], mlist[2,2], mlist[2,3])
```

```
1384.487 706.38 15.105999999999998 0.030149626863362672
1394.061 921.854 15.921 0.0304138126514911
724.86 986.57 14.02 0.03001666203960727
```

In [6]:

```
mag = mlist[:, 2]
merr = mlist[:, 3]
```

[:, 2]は2番目のインデックスが2のものを全て取り出すという意味です。このようにして、特定の列だけを抽出して、magおよびmerrの配列に保存しておきます。

In [7]:

```
mag[:10] # ここでは、長いので最初の10コだけ抽出
```

Out[7]:

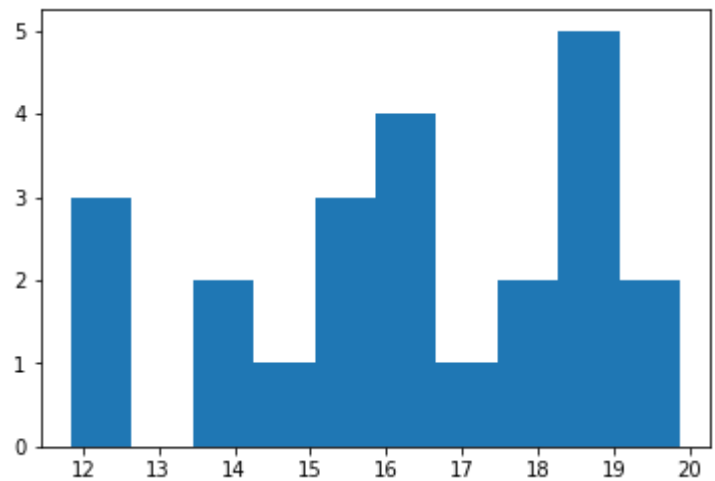
```
array([15.106, 15.921, 14.02 , 15.182, 18.2  , 18.561, 15.943, 19.05 ,
       18.303, 18.906])
```

光度関数のヒストグラム

まずは何も考えずに等級のヒストグラムを描いてみます。**plt.hist()**を使います。

In [8]:

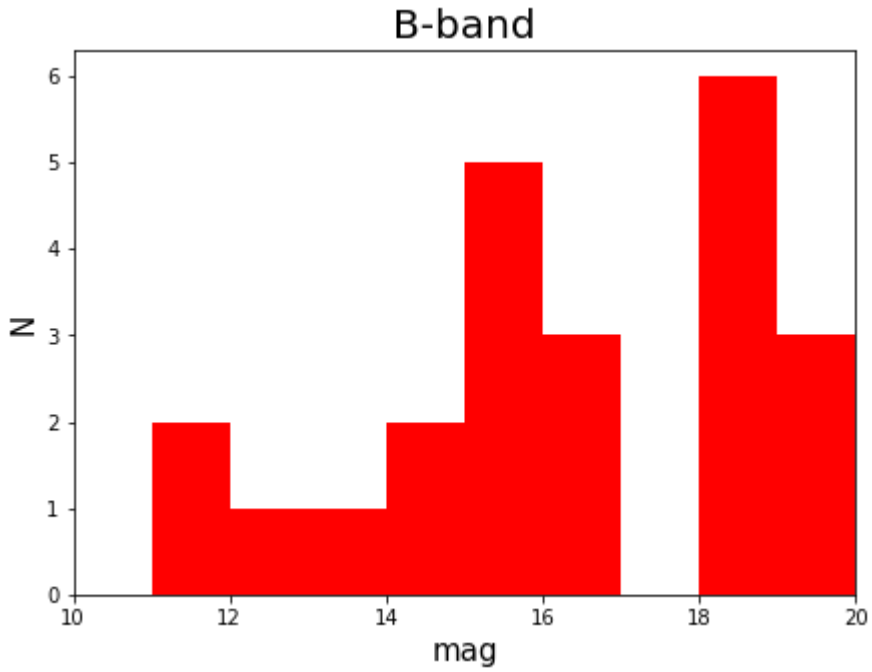
```
plt.hist(mag)
plt.show()
```



次に、オプションをいくつか加えてみます。

In [9]:

```
plt.figure(figsize=(7, 5))  # 図のサイズ
plt.hist(mag, bins=10, range=(10,20), color='red')  #ビンの数、ヒストグラムの範囲、色を指定
plt.xlim(10, 20) #グラフのX軸の範囲
plt.title('B-band', fontsize=20)
plt.xlabel('mag', fontsize=15)
plt.ylabel('N', fontsize=15)
plt.show()
```



関数でどんな引数が使えるかを調べたい時には、下のように?をおしりにつけます。

In [10]:

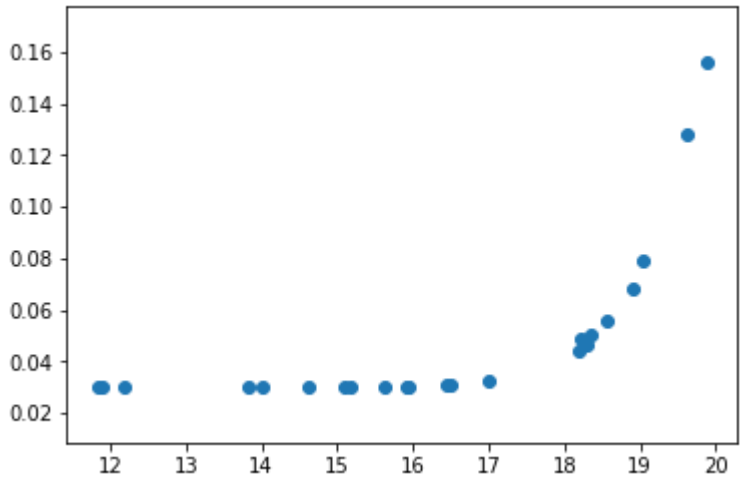
```
plt.hist?
```

等級 vs. 等級エラープロット

まずは何も考えずにプロット。**plt.scatter()**を使います。

In [11]:

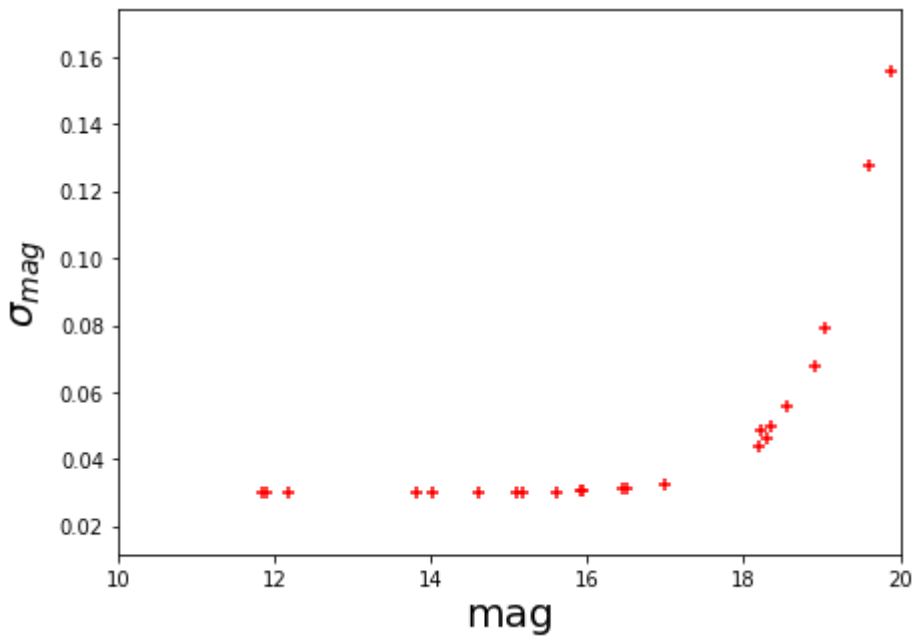
```
plt.scatter(mag, merr)
plt.show()
```



これも、いくつかオプションを加えてみます。

In [12]:

```
plt.figure(figsize=(7, 5))
plt.scatter(mag, merr, marker='+', color='red')
plt.xlim(10, 20)
plt.xlabel('mag', fontsize=20)
plt.ylabel('$\sigma_{mag}$', fontsize=20) # TeXの表記がつかえます
plt.show()
```



FITS画像の表示

FITS画像をnotebook内に表示します。
そのためには、astropy.io.fitsでデータをndarrayとして読み込む必要があります。

In [13]:

```
from astropy.io import fits
```

In [14]:

```
img = fits.getdata('btarget1n5trim.fits')
```

```
WARNING: VerifyWarning: Invalid 'BLANK' keyword in header.  The 'BLANK' keyword is on
ly applicable to integer data, and will be ignored in this HDU. [astropy.io.fits.hdu.image]
```

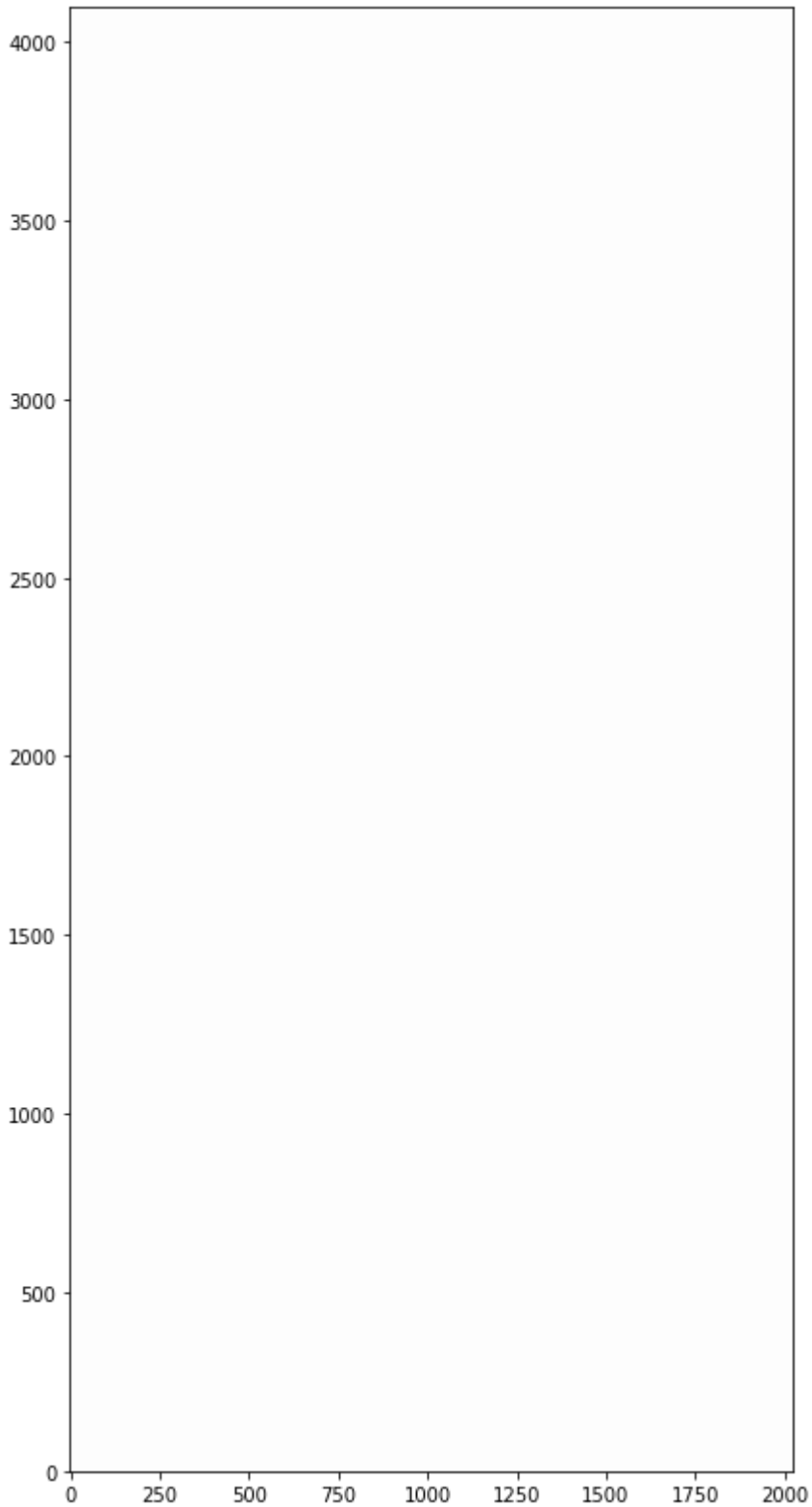
IRAFで処理したFITSヘッダにはBLANKというキーワードの行が残ってしいます。するとこのような警告が出ます。ここでは無視して大丈夫です。

まずは何も考えずに表示してみます。

In [15]:

```
plt.figure(figsize=(7, 14))
plt.imshow(img, plt.cm.gray, origin='lower', interpolation='none')

plt.show()
```



plt.cm.grayはカラーマップです。plt.cm.[カラーの名前]で指定します。
http://matplotlib.org/examples/color/colormaps_reference.html
(http://matplotlib.org/examples/color/colormaps_reference.html)
上のままではよくわかりません。表示レベルをちゃんと設定してやります。

講習3で、btarget1n5trim.fitsのバックグラウンドのメジアンが69でばらつきが7.4であることを求めました。 これをもとに表示レベルを設定します。

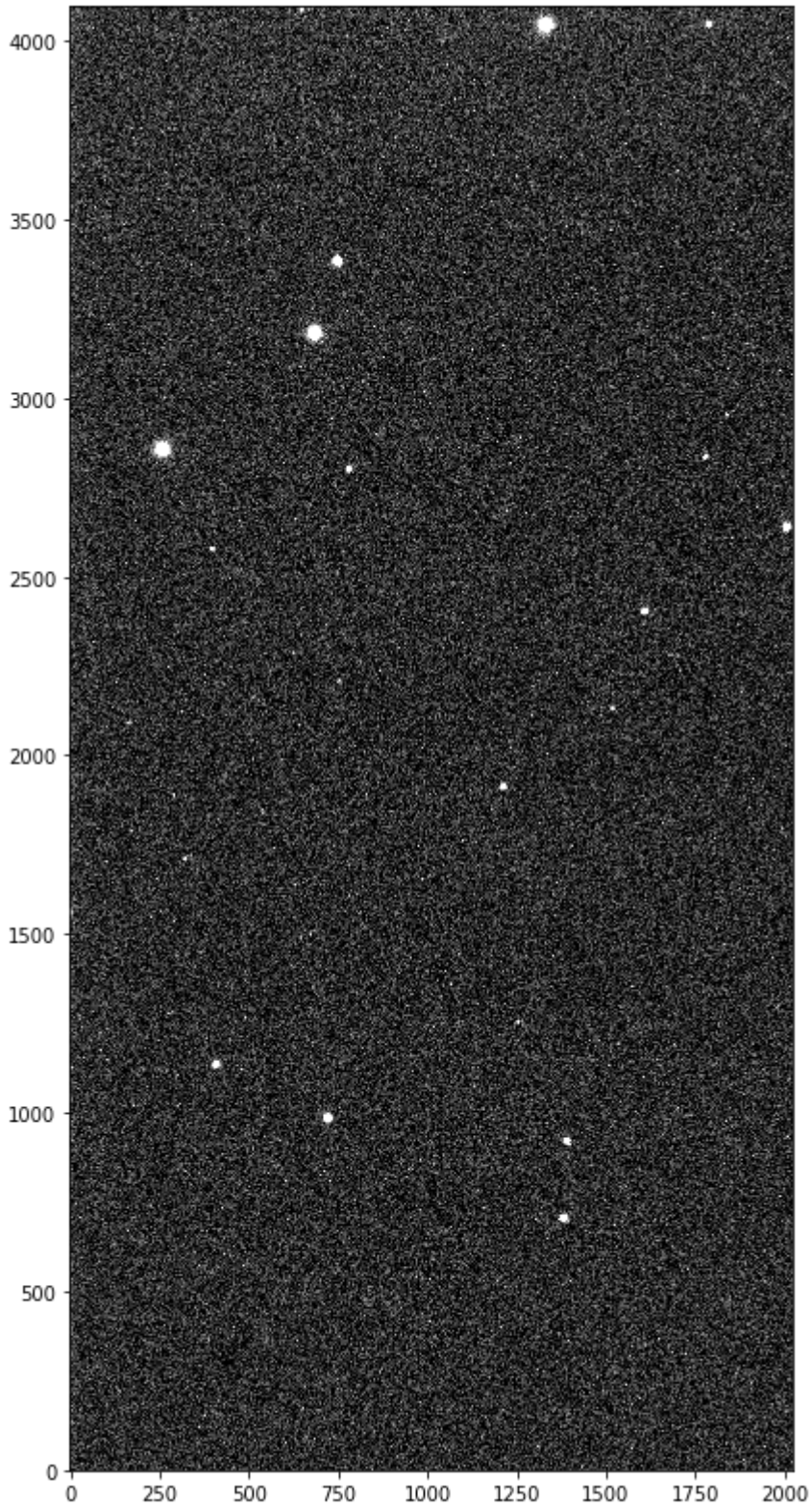
In [16]:

```
med = 69
std = 7.4
```

表示レベルの最小と最大(vminとvmax)をmed - std, および med + 5 stdに設定します。

In [17]:

```
plt.figure(figsize=(7, 14))
plt.imshow(img, plt.cm.gray, vmin=med - std, vmax = med + 5 * std, origin='lower', interpolation='none')
plt.show()
```



ここに印をいれてみます。 result1c.txtの中から最初の二つの星を選んでみます。

In [18]:

```
! head -2 result1c.txt
```

```
1384.487 706.380 15.105999999999998 0.030149626863362672
1394.061 921.854 15.921 0.0304138126514911
```

原点ピクセルのXY座標は、IRAFでは(1, 1)でPythonでは(0, 0)です。なので、下では座標値から1を引いてやります。

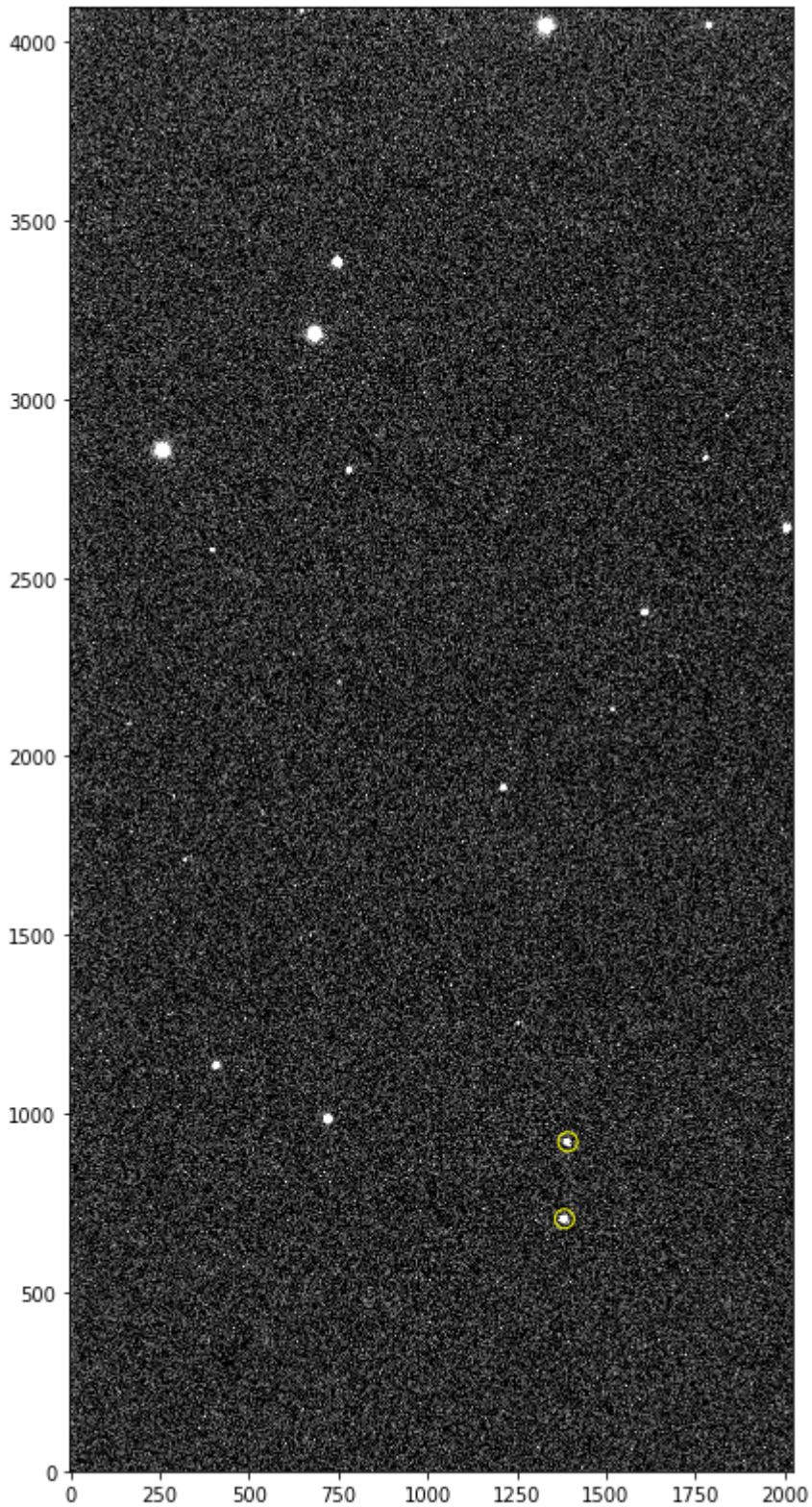
In [19]:

```
xcoo = [1383.5, 1393.1]
ycoo = [705.4, 920.9]
```

このxcoo, ycooをplt.scatter()でオーバープロットします。
xcoo, ycooのそれぞれの配列から順番にペアがプロットされます。等級vs等級エラーのときと同じです。
plt.scatter()のオプションのs=100は印の円の大きさです。面積で指定します。半径を倍にしたければ値を4倍にします。

In [20]:

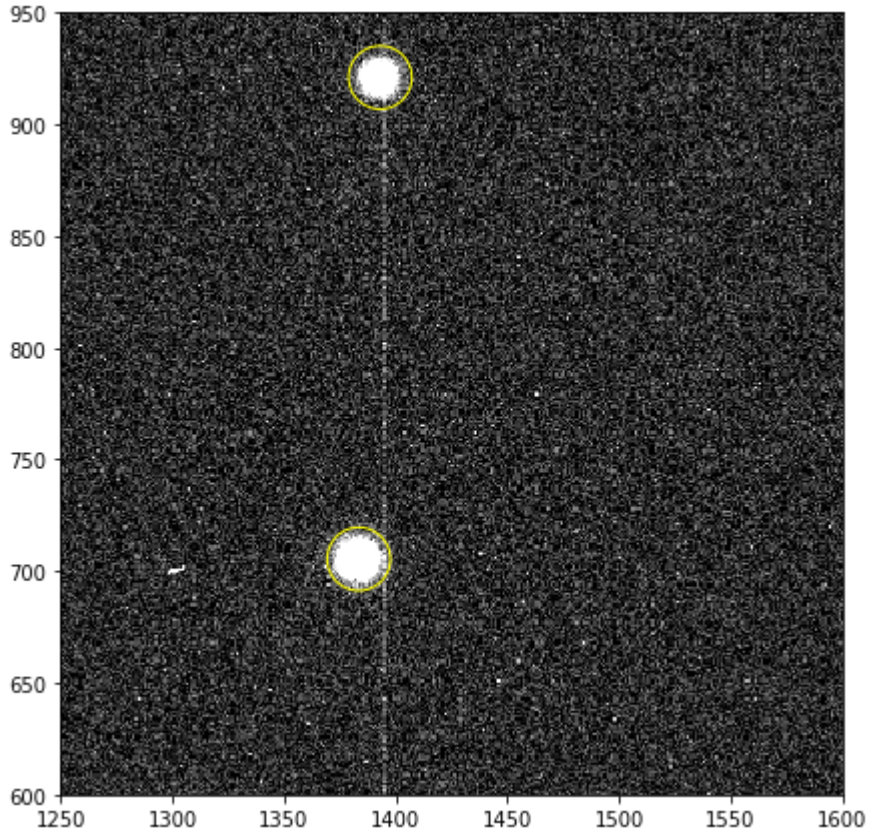
```
plt.figure(figsize=(7, 14))
plt.imshow(img, plt.cm.gray, vmin=med - std, vmax = med + 5 * std, origin='lower',
           interpolation='none')
plt.scatter(xcoo, ycoo, edgecolors='yellow', facecolors='none', s=100)
plt.show()
```



標準星の周辺だけ拡大します。plt.xlim()とplt.ylim()が加わっただけです。
あとは、拡大したので印の大きさも変えました。

In [21]:

```
plt.figure(figsize=(7, 14))
plt.imshow(img, plt.cm.gray, vmin=med - std, vmax = med + 5 * std, origin='lower',
           interpolation='none')
plt.scatter(xcoo, ycoo, edgecolors='yellow', facecolors='none', s=1000)
plt.xlim(1250, 1600)
plt.ylim(600, 950)
plt.show()
```



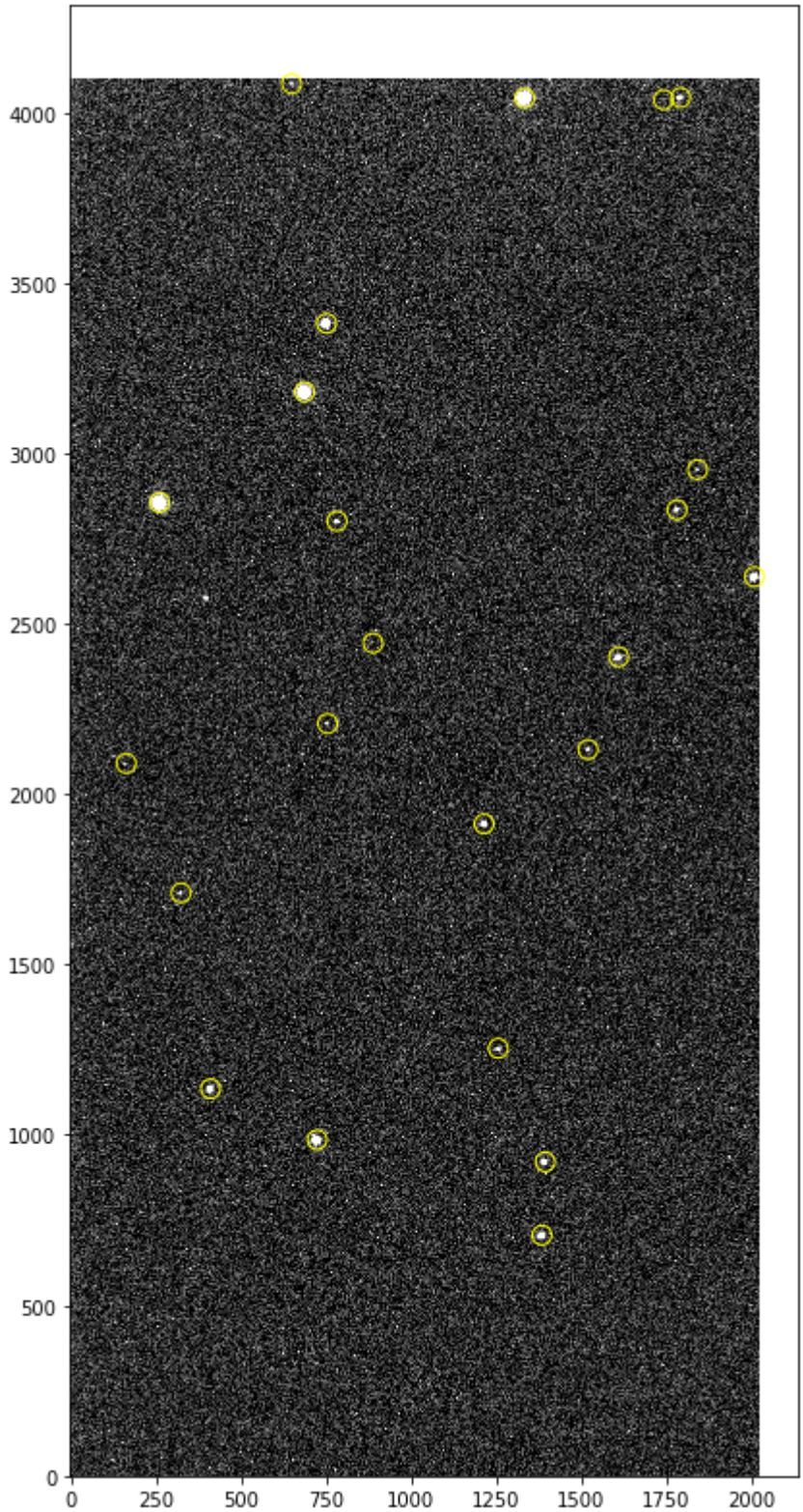
せっかくなので、測光した全ての星をプロットします。

In [22]:

```
xstar = mlist[:, 0] - 1
ystar= mlist[:, 1] -1
```

In [23]:

```
plt.figure(figsize=(7, 14))
plt.imshow(img, plt.cm.gray, vmin=med - std, vmax = med + 5 * std, origin='lower',
           interpolation='none')
plt.scatter(xstar, ystar, edgecolors='yellow', facecolors='none', s=100)
plt.show()
#plt.savefig('myfig.png')  # 上のplt.show()をコメントアウトし、この行を実行すると
                           # ファイルに保存できる。
```



演習6

演習3で行った'btarget2n5.fits'の測光結果を用いて、
(1) 「光度関数のヒストグラム」と 「等級vs等級エラーのプロット」を作成してください。
(2) FITS画像をnotebookに表示して、そこに測光した星をプロットしてください。

講習7 --- プログラムの使い回し ~ スクリプト作成等

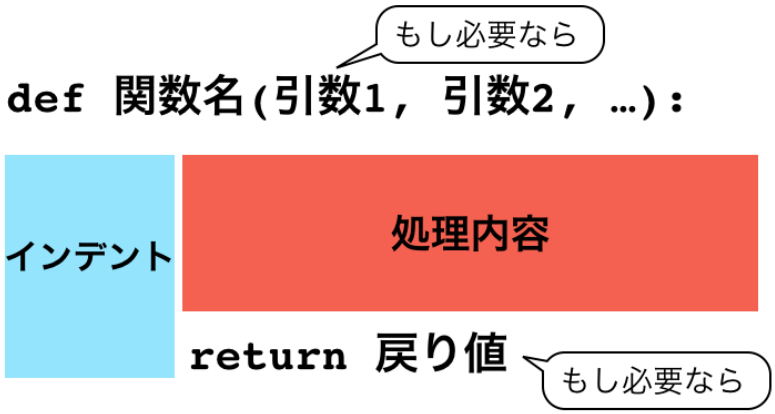
ここまで、jupyter notebookでインタラクティブに処理を行い、いろいろなコマンドの使い方、 簡単なpythonのプログラミングを見てきました。

次に、ひとかたまりのプログラムに汎用性をもたせるという観点で話をすすめます。

ここでは、まず、関数の定義の仕方から始め、pythonスクリプトの作成、コマンドライン引数の使い方、自作モジュールの使い方を説明します。

自作関数

ここまでの処理の中で、いくつかの処理の「かたまり」は、読み込むファイル名だけが変わって、繰り返し出てきました。
そのような「かたまり」は関数として定義しておくと効率的にコードを作成することができます。



バックグラウンドのメジアンとノイズを、3シグマクリップして評価する処理なんていうのは、よく使うので関数にしておくと便利です。

In [1]:

```
import numpy as np
from astropy.io import fits
```

In [2]:

```
def getbackground(infits):

    data = fits.getdata(infits)
    med = np.median(data)
    std = np.std(data)

    for i in range(5):
        xx = np.where((data > med - 3 * std) & (data < med + 3 * std))
        med = np.median(data[xx])
        std = np.std(data[xx])

    return med, std
```

In [3]:

```
med, std = getbackground('sample/btarget1n5small.fits')
print(med, std)
```

```
69.305275 7.4028854
```

FITSを表示する処理もよく使うでしょう。関数にしておきます。
ここでは、getbackground( )も中で使っています。

In [4]:

```
import matplotlib.pyplot as plt
```

In [5]:

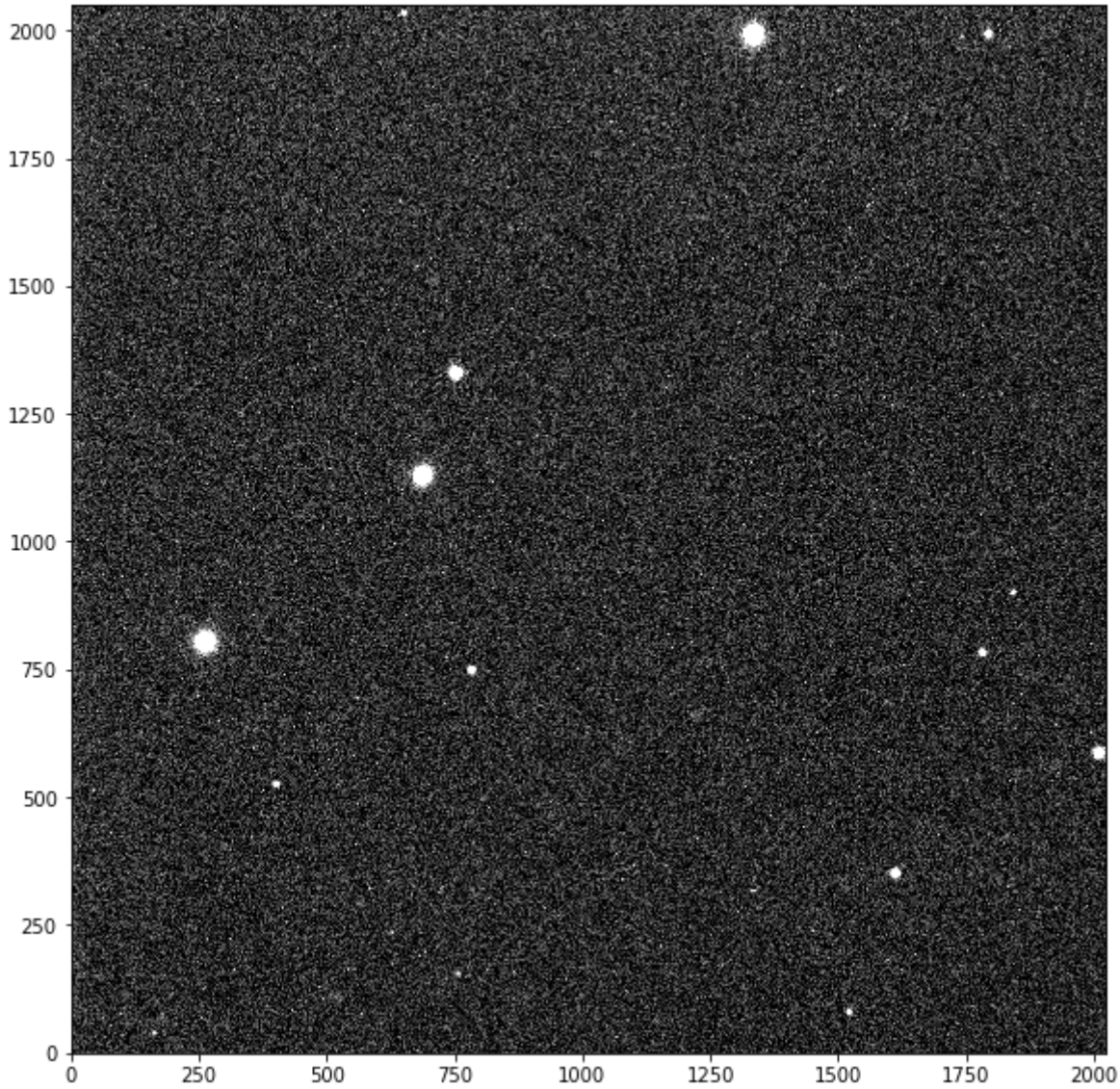
```
def showfits(infits):

    med, std = getbackground(infits)
    imdata = fits.getdata(infits)

    plt.figure(figsize=(10, 10))
    plt.imshow(imdata, plt.cm.gray, vmin=med - std, vmax = med + 5 * std, origin='lower',
               interpolation='none')
    plt.show()
```

In [6]:

```
showfits('sample/btarget1n5small.fits')
```



pythonスクリプトを作成

さて、notebookから飛び出して、pythonスクリプトを作成しましょう。
pythonスクリプトを作成するのは難しいことではありません。
テキストエディタで、セルに書き込んでいた内容を下記ならべ、拡張子が.py のファイルとして保存すればよいのです。
さらに、unix/linux系ではファイル冒頭に
#!/usr/local/bin/python3.5
のようにpythonのpathを書き込むのが一般的です。

In [7]:

```
import numpy as np

data1 = np.array([0, 1, 2])
data2 = np.array([[0, 0, 0], [10, 10, 10], [20, 20, 20], [30, 30, 30]])

data3 = data2 + data1

print(data3)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

これを、mycode.pyというファイルに書き込みましょう。

なんでもお好みのテキストエディタを使ってください。あるいは、jupyter notebookにはテキストエディタ機能もあります。
jupyter notebook起動画面の右上のNew > Text File を選ぶと新規テキストファイルの画面になります。

そのうえで、(chmod +x してから)コマンドラインで実行します。あるいは下のようにcellからも実行できます。

In [8]:

```
%run mycode.py
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

コマンドライン引数

スクリプトに書き出したのであれば、コマンドライン引数も使いたくなります。

In [9]:

```
targetfits = 'sample/btarget1n5small.fits'
```

In [10]:

```
from astropy.io import fits
import numpy as np

data = fits.getdata(targetfits)
med = np.median(data)
std = np.std(data)

for i in range(5):
    xx = np.where((data > med - 3 * std) & (data < med + 3 * std))
    med = np.median(data[xx])
    std = np.std(data[xx])
    print ('{:.2f} {:.2f}'.format(med, std))
```

```
69.33 9.52
69.31 7.56
69.31 7.42
69.31 7.41
69.31 7.40
```

ここで、targetfitsをコマンドライン引数として読み込めれば、いろんなFITSファイルに対してこのプログラムが使えます。

コマンドライン引数を取り込むには

```
import sys
sys.argv
```

を使います。
コマンドラインに入力された文字列が空白で区切られて、リスト sys.argv に格納されます。
sys.argv[0] はプログラム名そのものです。1番目の引数はsys.argv[1]です。

In [13]:

```
cat getbackground.py
```

```
import sys
import numpy as np
from astropy.io import fits

data = fits.getdata(sys.argv[1])
med = np.median(data)
std = np.std(data)

for i in range(5):
    xx = np.where((data > med - 3 * std) & (data < med + 3 * std))
    med = np.median(data[xx])
    std = np.std(data[xx])
    print ('{:.2f} {:.2f}'.format(med, std))
```

In [14]:

```
%run getbackground.py sample/btarget1n5small.fits
```

```
69.33 9.52
69.31 7.56
69.31 7.42
69.31 7.41
69.31 7.40
```

モジュール作成

よく使う処理を、notebookの中で関数として定義して使用するのは楽チンなのですが、
notebookファイルを作成するたびに、前のnotebookからコピペして使うのはちょっと面倒ですね。

頻繁に使う自作の関数はモジュールにしておきましょう。

/home/nakajima/mypylib/mymodule.py のようなファイルを作成します。
必要なモジュールを冒頭でimportしておき、あとは自前の関数をどんどん書き込んでいけばよいです。
そして、他のプログラムから使うときには、

```
import sys
sys.path.append('/home/nakajima/mypylib/')
```

として、pathを通しておき、

```
import mymodule
```

を宣言します。 mymodule.pyの.pyはimportで呼ぶときには不要です。

ここでは、このディレクトリの中に'mypylib'というディレクトリを作成し、その中にmymodule.pyを作りました。

In [15]:

```
cat mypylib/mymodule.py
```

```
import numpy as np
from astropy.io import fits
import matplotlib.pyplot as plt

def getbackground(infits):

    data = fits.getdata(infits)
    med = np.median(data)
    std = np.std(data)

    for i in range(5):
        xx = np.where((data > med - 3 * std) & (data < med + 3 * std))
        med = np.median(data[xx])
        std = np.std(data[xx])

    return med, std


def showfits(infits):

    med, std = getbackground(infits)
    imdata = fits.getdata(infits)

    plt.figure(figsize=(10, 10))
    plt.imshow(imdata, plt.cm.gray, vmin=med - std, vmax = med + 5 * std, origin='lower',
interpolation='none')
    plt.show()
```

次々とこの続きに、自分の関数を書き込んでいけばよいです。

さて、ここの例では次のようにしてpathを通し、

In [16]:

```
import sys
sys.path.append('./mypylib/')
```

mymoduleをimportして、mymodule.getbackground( ) を使います。

In [17]:

```
import mymodule
```

In [18]:

```
med, std = mymodule.getbackground('sample/btarget1n5small.fits')
print(med, std)
```

```
69.305275 7.4028854
```

余談ですが、私は、/home/nakajima/mynotebook/ のようなディレクトリを作成し、その中に.ipynbファイルをまとめています。
データのあるディレクトリには、notebookの中で cd で change directoryして移動して処理を行います。

まとまっていると何がいいかというと、検索できるんですね。
ipynbファイルはJSONという形式のテキストファイルです。 grepで検索できます。

たくさんたまってくると、あれどこでやったかなあ。。。なんてこともよくあります。