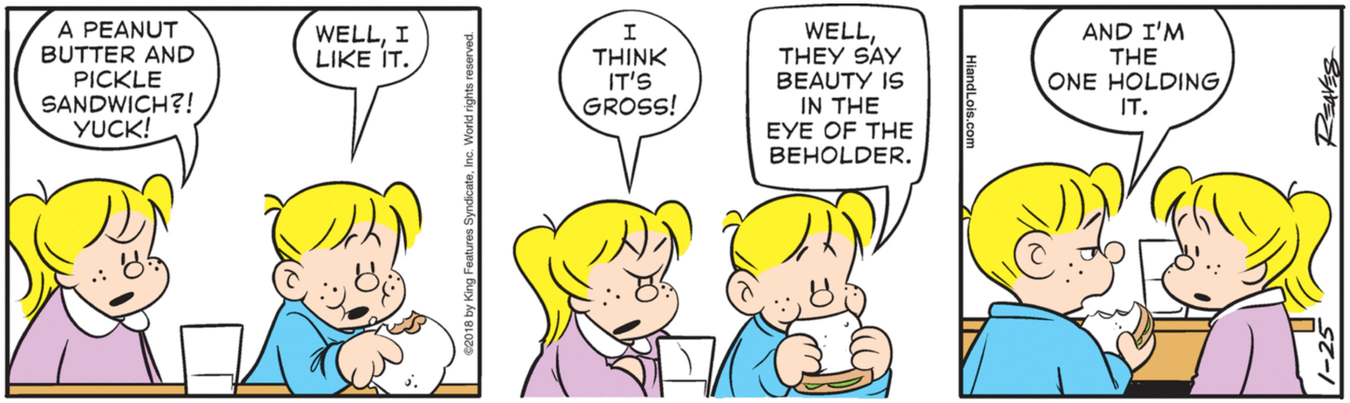
Implementation Hints

Coding Style

CU Boulder

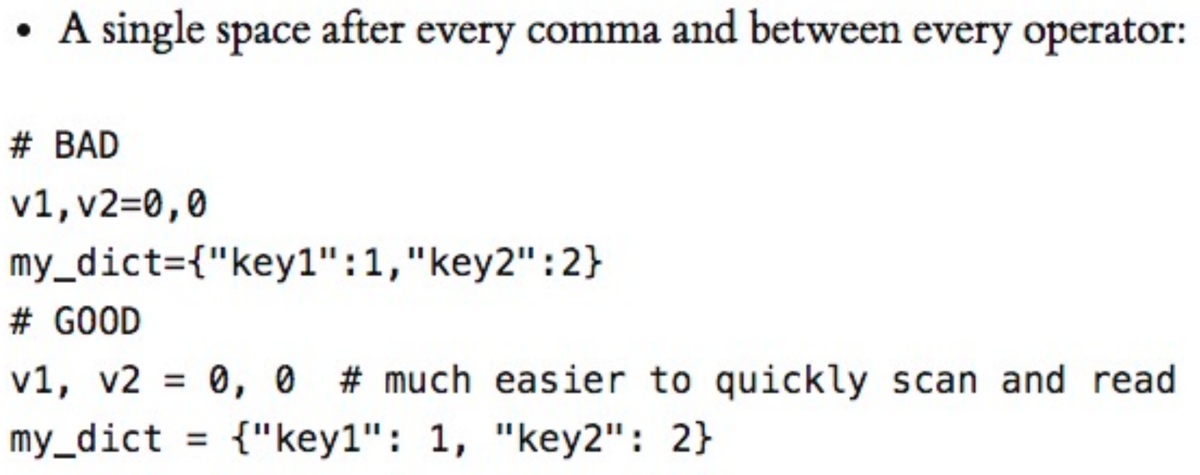
- **Is good code really just in the eye of the beholder?**


A PEANUT BUTTER AND PICKLE SANDWICH?! YUCK!
WELL, I LIKE IT.
I THINK IT'S GROSS!
WELL, THEY SAY BEAUTY IS IN THE EYE OF THE BEHOLDER.
AND I'M THE ONE HOLDING IT.
HiandLois.com
REAVES
1-25
©2018 by King Features Syndicate, Inc. World rights reserved.

DO YOU KNOW WHO COINED THE PHRASE 'BEAUTY IS IN THE EYE OF THE BEHOLDER'?
I'M GUESSING IT WAS WHOEVER TRAINED YOU TO BE A SCULPTOR.
1-30-17
Dist. by Creators
JohnHartStudios.com

Good Code

- **There is often a python coding idiom, or “pythonic” way to do things**
- **A link with some useful guidelines**
  - https://blog.hartleybrody.com/python-style-guide/
  - Example (idiom: readability matters):
    - ✓ Code is read more times than it is written (and by different people)

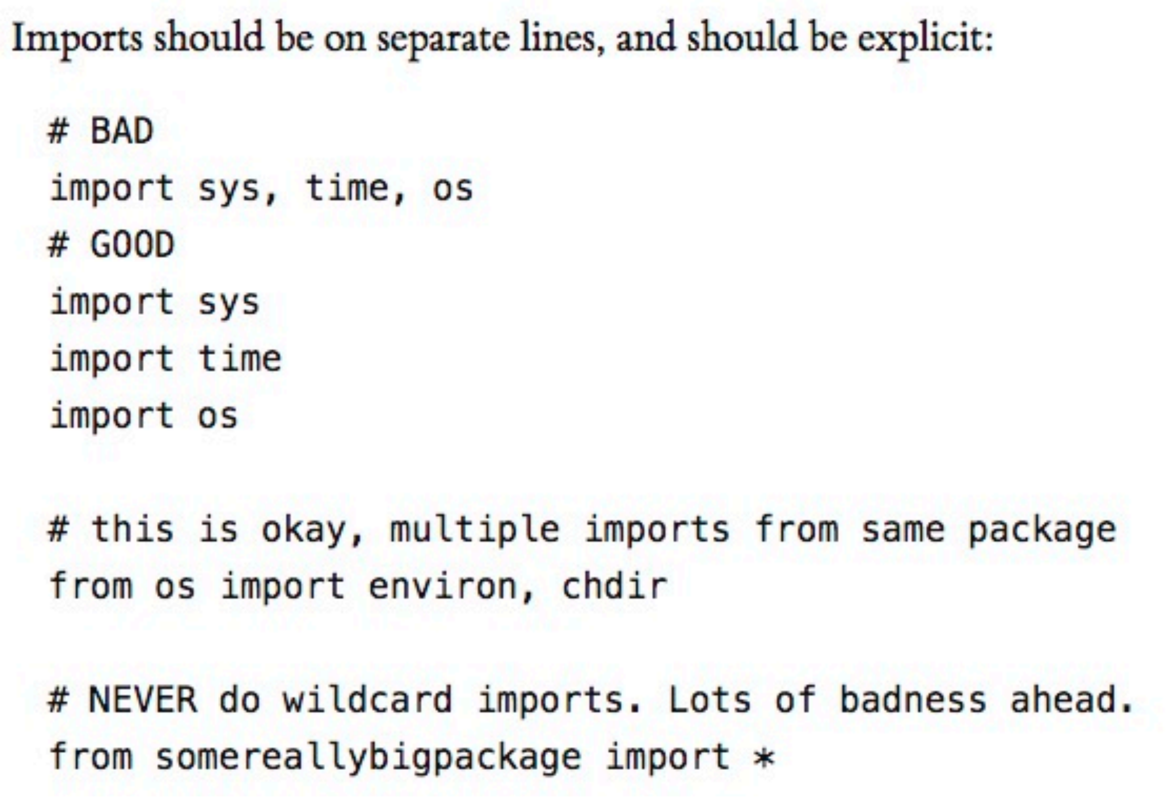


- A single space after every comma and between every operator:

```
# BAD
v1,v2=0,0
my_dict={"key1":1,"key2":2}
# GOOD
v1, v2 = 0, 0  # much easier to quickly scan and read
my_dict = {"key1": 1, "key2": 2}
```

Good Code

- **More**

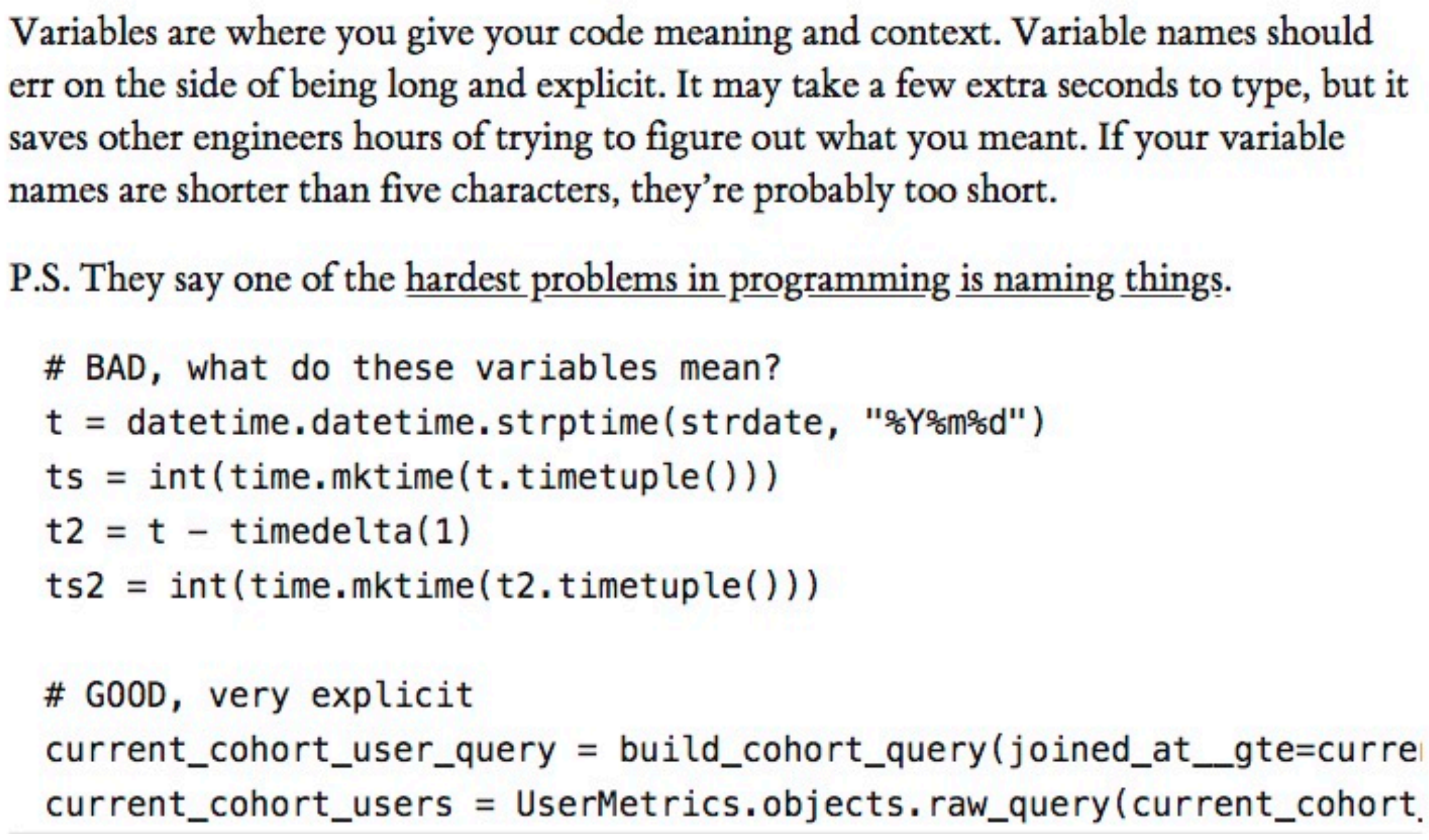


Imports should be on separate lines, and should be explicit:

```
# BAD
import sys, time, os
# GOOD
import sys
import time
import os

# this is okay, multiple imports from same package
from os import environ, chdir

# NEVER do wildcard imports. Lots of badness ahead.
from somereallybigpackage import *
```

Good Code

- **More**



Variables are where you give your code meaning and context. Variable names should err on the side of being long and explicit. It may take a few extra seconds to type, but it saves other engineers hours of trying to figure out what you meant. If your variable names are shorter than five characters, they're probably too short.

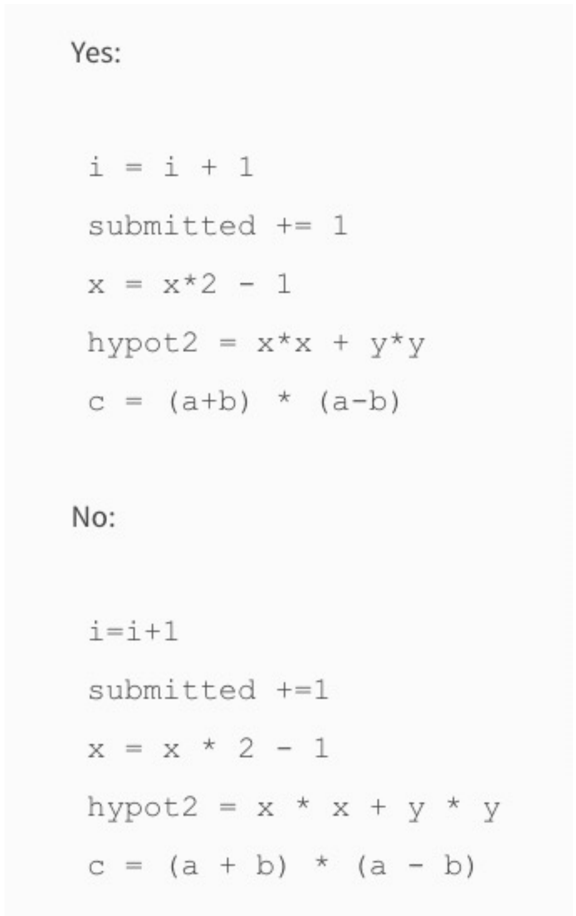
P.S. They say one of the hardest problems in programming is naming things.

```
# BAD, what do these variables mean?
t = datetime.datetime.strptime(strdate, "%Y%m%d")
ts = int(time.mktime(t.timetuple()))
t2 = t - timedelta(1)
ts2 = int(time.mktime(t2.timetuple()))

# GOOD, very explicit
current_cohort_user_query = build_cohort_query(joined_at__gte=curre
current_cohort_users = UserMetrics.objects.raw_query(current_cohort_
```

Good Code

CU Boulder

- **Another good resource**
  - https://www.python.org/dev/peps/pep-0008/
  - Examples (readability):



Yes:

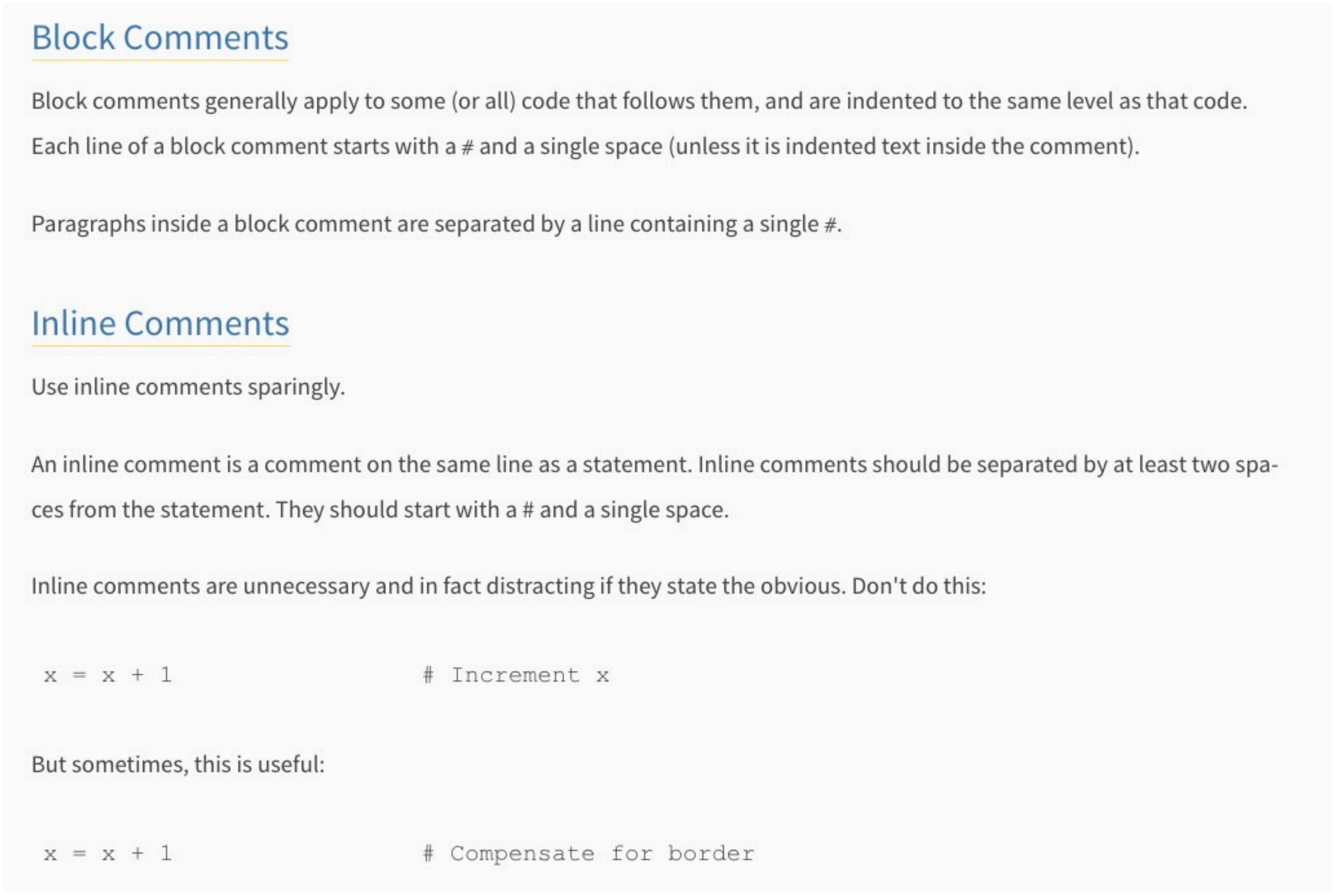
```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Good Code

- **More**



Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spa-ces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

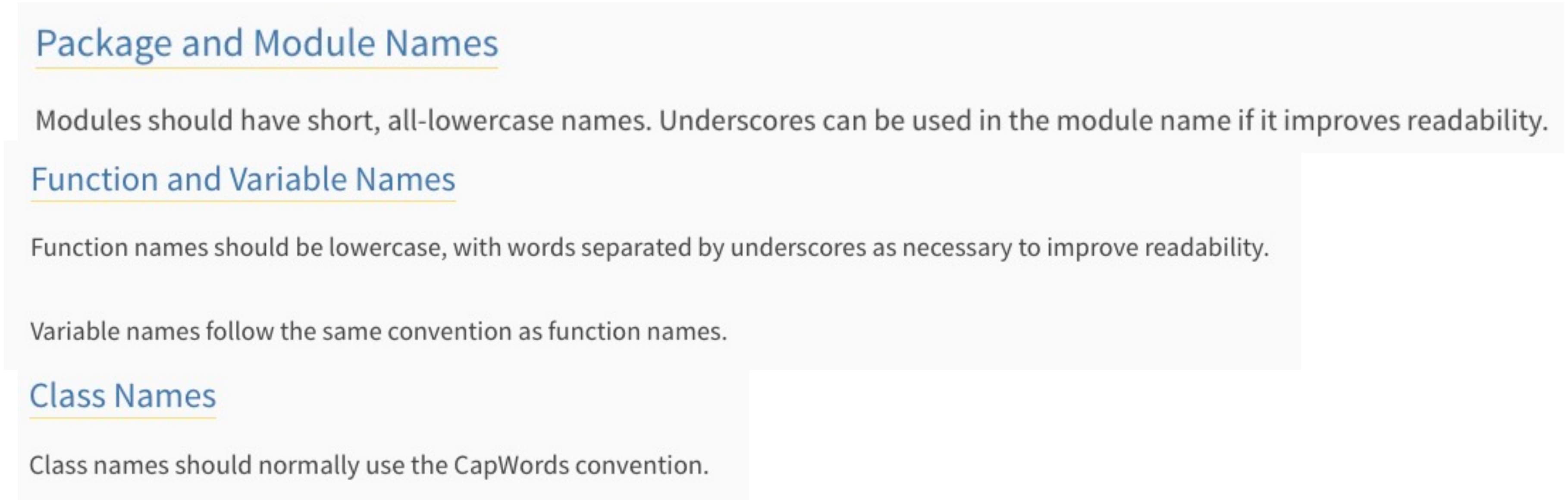
But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

Good DSP Code with Numpy

CU Boulder

- **Two common naming conventions**
  - ”snake case”: day_off
  - “camel case or CapWords”: DayOff
- **Module names**
  - In this course a module is a file with multiple functions



Package and Module Names

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability.

Function and Variable Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

Class Names

Class names should normally use the CapWords convention.

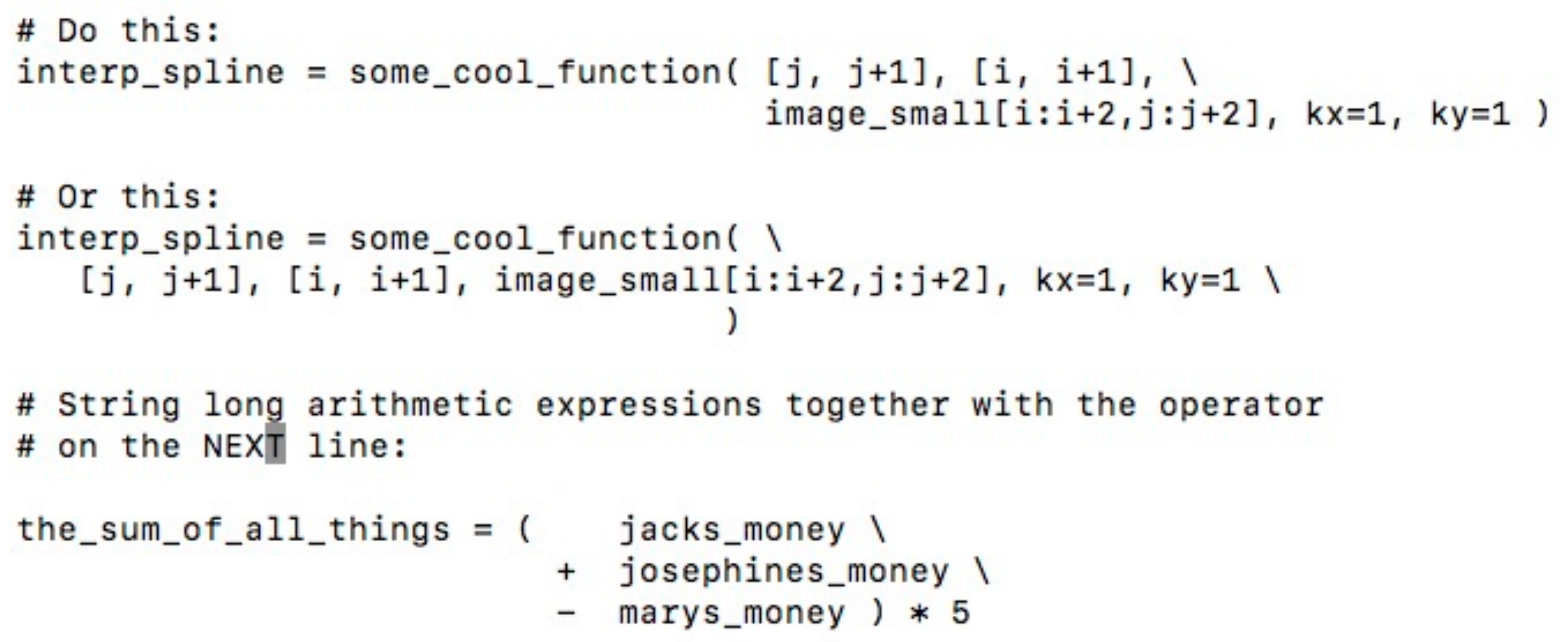
Good DSP Code with Numpy

- **Variable naming**
  - Longer names are often better than shorter names
    - ✓ However, if a variable will be used a lot in complicated expressions, a shorter name might be used anyway
      - – You might put a comment in your code somewhere to say what the short variable name is
  - Use the variables i, j, k, m, n to represent for( ) loop counters. Avoid them for floating point variable names
  - Make an effort to align delimiters when expressions span more than one line. Example

Good DSP Code with Numpy

CU Boulder

- **Make an effort to align syntax elements when expressions span more than one line**



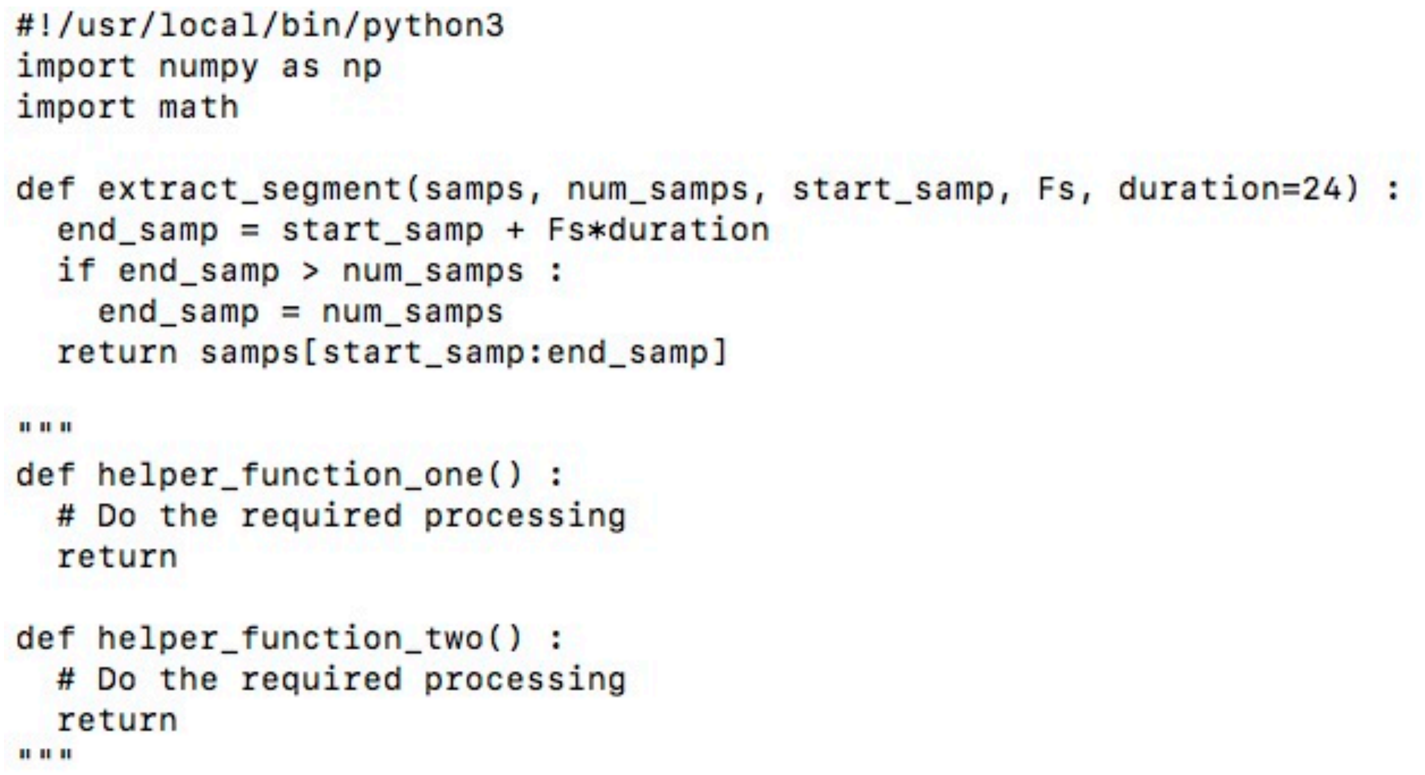
```
# Do this:
interp_spline = some_cool_function( [j, j+1], [i, i+1], \
                                    image_small[i:i+2,j:j+2], kx=1, ky=1 )

# Or this:
interp_spline = some_cool_function( \
   [j, j+1], [i, i+1], image_small[i:i+2,j:j+2], kx=1, ky=1 \
                                  )

# String long arithmetic expressions together with the operator
# on the NEXT line:

the_sum_of_all_things = (    jacks_money \
                          +  josephines_money \
                          -  marys_money ) * 5
```

Good DSP Code with Numpy

- **There is usually a “natural” chunk size for the algorithm you are writing**
  - A frame of audio
  - A block of a picture
- **When you use a for loop, try to loop over the natural chunk size**
  - Often numpy functions allow a chunk to be efficiently processed with a single function call
- **Avoid for( ) loops that touch every sample (e.g. audio sample or image pixel) if possible**
  - Slow to execute

Good DSP Code with Numpy

- **Example software architecture for lab one**
  - Can create a single module, lab1_funcs.py, to contain the helper functions you will need

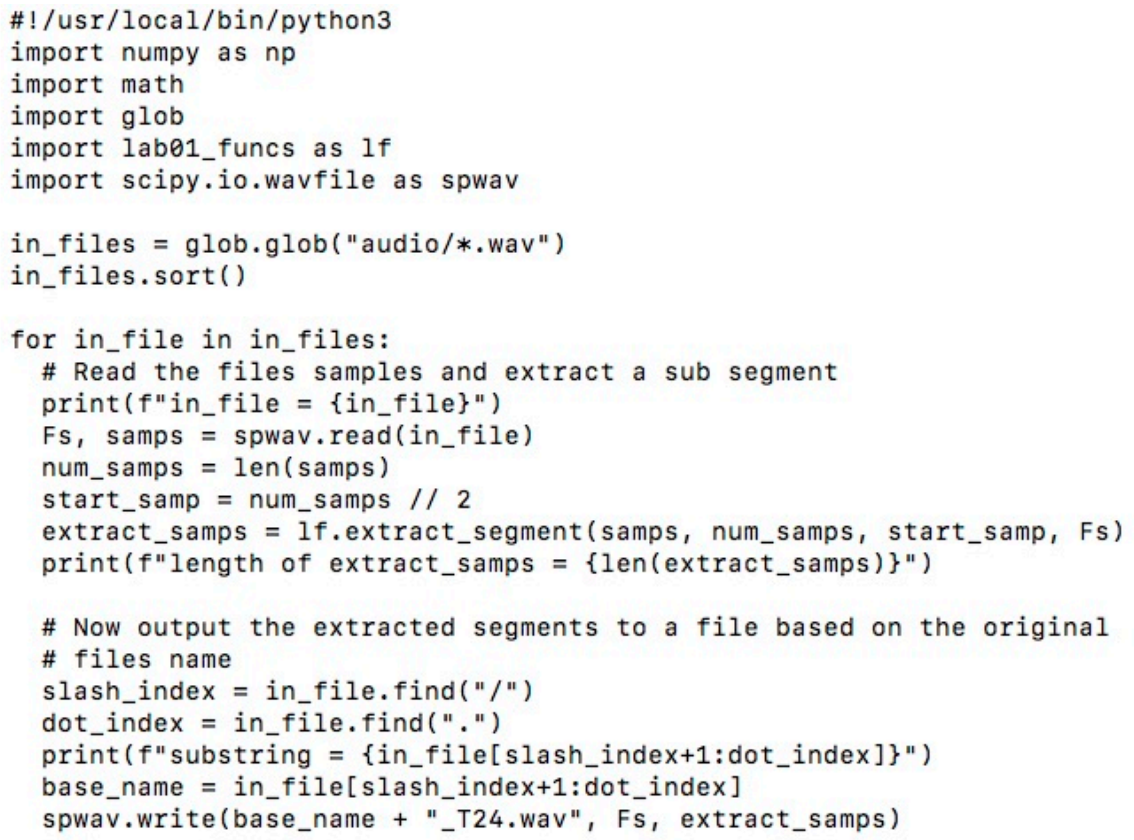


```
#!/usr/local/bin/python3
import numpy as np
import math

def extract_segment(samps, num_samps, start_samp, Fs, duration=24) :
  end_samp = start_samp + Fs*duration
  if end_samp > num_samps :
    end_samp = num_samps
  return samps[start_samp:end_samp]

"""
def helper_function_one() :
  # Do the required processing
  return

def helper_function_two() :
  # Do the required processing
  return
"""
```

Good DSP Code with Numpy

CU Boulder

- **continued...**
  - Can then put your code for each problem in a separate file, for example, lab01_a02.py



```
#!/usr/local/bin/python3
import numpy as np
import math
import glob
import lab01_funcs as lf
import scipy.io.wavfile as spwav

in_files = glob.glob("audio/*.wav")
in_files.sort()

for in_file in in_files:
  # Read the files samples and extract a sub segment
  print(f"in_file = {in_file}")
  Fs, samps = spwav.read(in_file)
  num_samps = len(samps)
  start_samp = num_samps // 2
  extract_samps = lf.extract_segment(samps, num_samps, start_samp, Fs)
  print(f"length of extract_samps = {len(extract_samps)}")

  # Now output the extracted segments to a file based on the original
  # files name
  slash_index = in_file.find("/")
  dot_index = in_file.find(".")
  print(f"substring = {in_file[slash_index+1:dot_index]}")
  base_name = in_file[slash_index+1:dot_index]
  spwav.write(base_name + "_T24.wav", Fs, extract_samps)
```

Good DSP Code with Numpy

CU Boulder

- **Core of computing loudness**

```
Fs, samps = spwav.read(in_file)
tmp_dat = np.reshape(samps, (-1,512))
loudness = np.zeros(num_frames, np.float)
for i in range(num_frames) :
  loudness[i] = np.std( tmp_dat[i] )
```

- **Core of computing ZCR**

```
zcr = np.zeros(num_frames, np.float)
for i in range(num_frames) :
  zcr[i] = np.sum(                                                   \
   0.5 * np.abs( np.sign(tmpdat[i,1:]) - np.sign(tmpdat[i,:-1]) )    \
                )                                                    \
         / (N -1)
```

Lab 2 Hints

Lab 2 hint

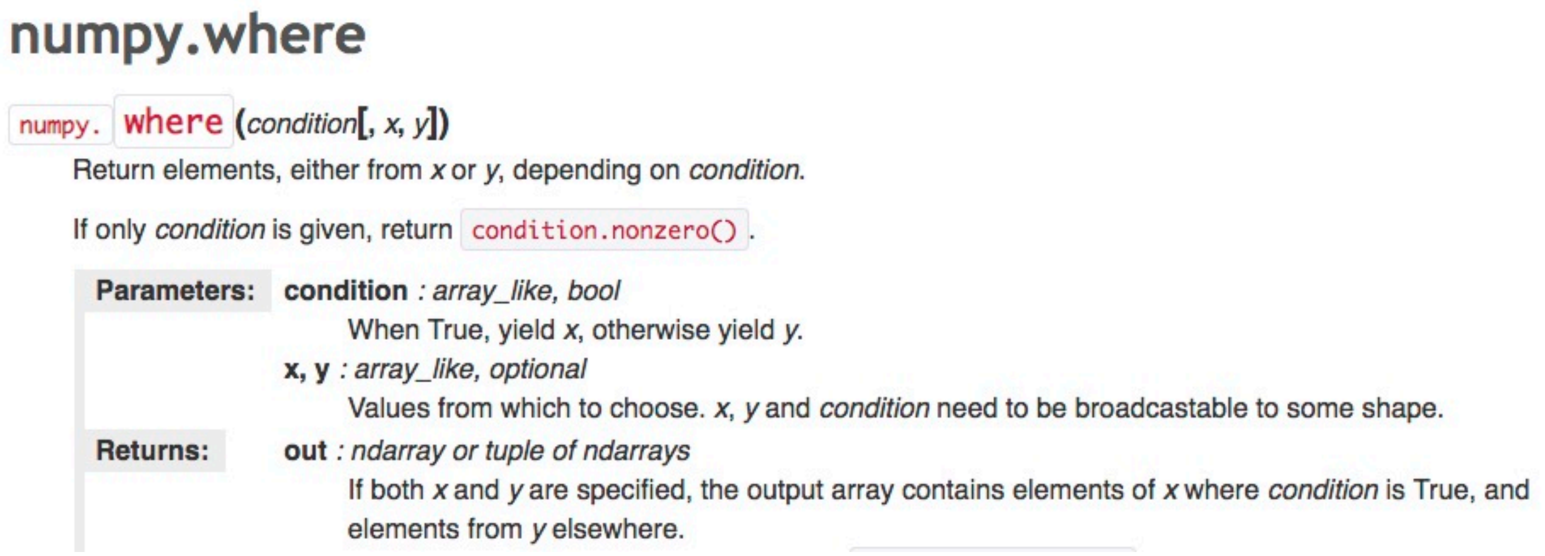
- **Can convert a numpy array to a new type like this**

```
image_out = image_out.astype( np.uint8 )
```

- **Can get the “shape” and dimenstions of an array (e.g. the rows and cols of an image) via**
  - array_name.shape
  - rows = array_name.shape[0]
  - cols = array_name.shape[1]

Lab 2 hint

- **numpy.where( ) is useful for the histogram and edge detection problems**



numpy.where

`numpy.where(condition[, x, y])`

Return elements, either from *x* or *y*, depending on *condition*.

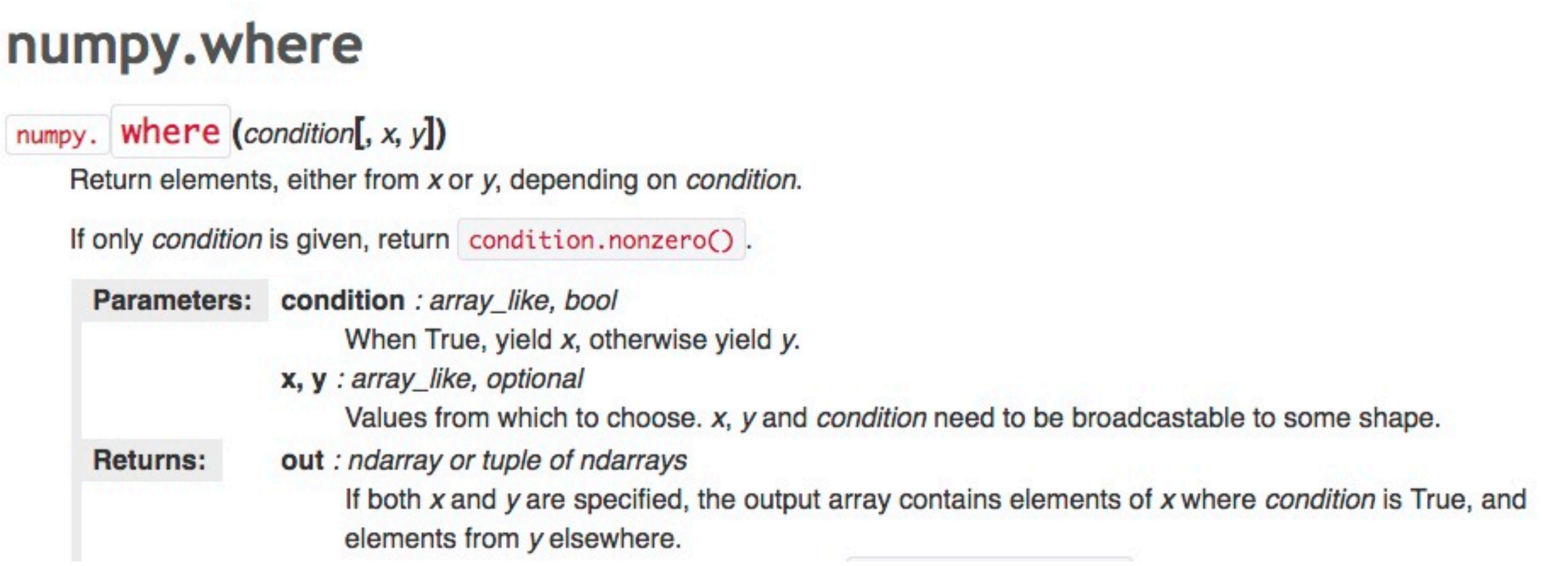
If only *condition* is given, return `condition.nonzero()`.

Parameters:	**condition** : *array_like, bool* When True, yield *x*, otherwise yield *y*. **x, y** : *array_like, optional* Values from which to choose. *x*, *y* and *condition* need to be broadcastable to some shape.
Returns:	**out** : *ndarray or tuple of ndarrays* If both *x* and *y* are specified, the output array contains elements of *x* where *condition* is True, and elements from *y* elsewhere.

Condition, for example, can be "*array==value*"

Lab 2 hint

- **And for the edge detector**



numpy.where

`numpy.where`(*condition*[, *x*, *y*])

Return elements, either from *x* or *y*, depending on *condition*.

If only *condition* is given, return `condition.nonzero()`.

Parameters:	**condition** : *array_like, bool* When True, yield *x*, otherwise yield *y*. **x, y** : *array_like, optional* Values from which to choose. *x*, *y* and *condition* need to be broadcastable to some shape.
Returns:	**out** : *ndarray or tuple of ndarrays* If both *x* and *y* are specified, the output array contains elements of *x* where *condition* is True, and elements from *y* elsewhere.

```
image_out = np.where(image_out < thresh, 0, 255)
```

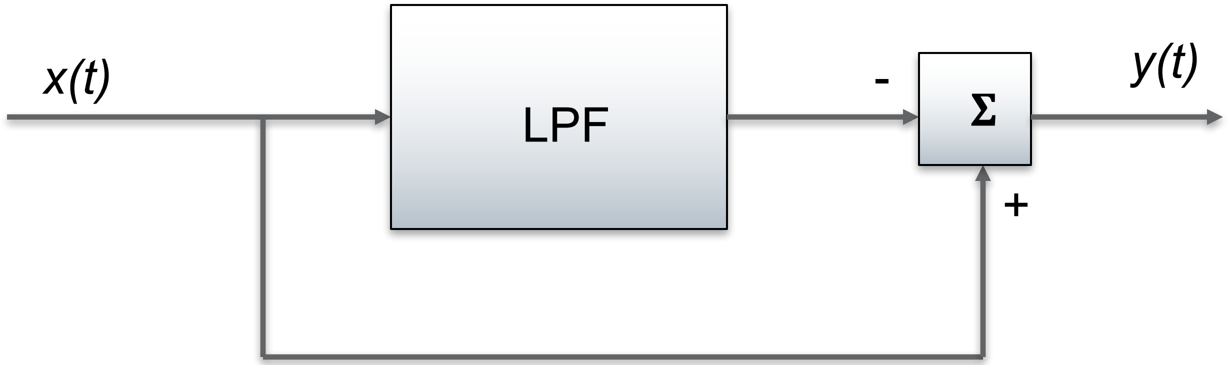
Lab 2 hint

- Bi-linear interpolation as shown in the example contained in the lab writeup will be useful to you

Lab 2 hint

CU Boulder

- **What does the following system do**

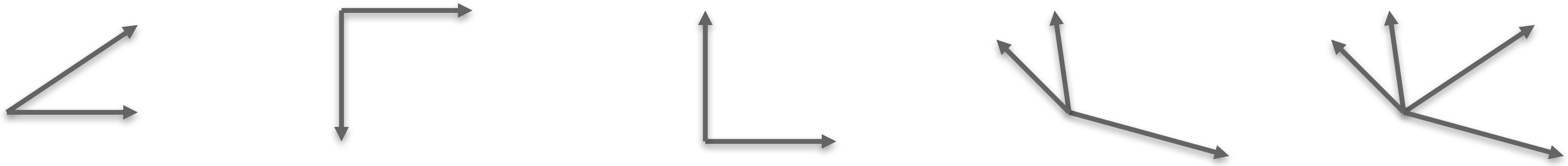


Least Squares Problem: Linear Regression

Span of a set of vectors

CU Boulder

- **The <u>span</u> of a set of vectors is the set of all vectors that can be reached by a linear combination of the vectors in the set**
  - All 5 of these sets span the plane ($\mathbb{R}^2$)



Column Space

- **Let A be an m-by-n matrix**
  - The column space of A is the set of m-by-1 vectors spanned by the columns. In other words
    - ✓ The set of vectors that can be "reached" by multiplying A$\underline{x}$ for some $\underline{\mathbf{x}} \in \mathbb{R}^n$
    - ✓ IMPORTANT: A$\underline{x}$ = $\underline{b}$ is exactly solvable only if $\underline{b}$ lies in the column space of A

Why does this matter?

- **It gives insight into solving systems of linear equations**
  - Let A be an N-by-N matrix

$$A \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix}$$

  - This has a solution only if **c** lies in the column space of A
    - ✓ If **c** is outside the column space, it cannot be reached, and there is no exact solution

General Case

- **Example: The matrix A might not be square. For example, it could represent an *over-determined* set of equations**
  - There are <u>more</u> equations than variables
  - A is now an m-by-n matrix with m > n
    - ✓ Since A is no longer square, it cannot have a proper matrix inverse
  - This problem frequently arises when many measurements are performed and used to estimate a small number of parameters (i.e. a regression is performed)

The Pseudo Inverse

- If $\underline{\mathbf{c}}$ is not in the column space of A, we instead seek an $\underline{\mathbf{x}}$ that minimizes the Mean Squared Error (MSE)

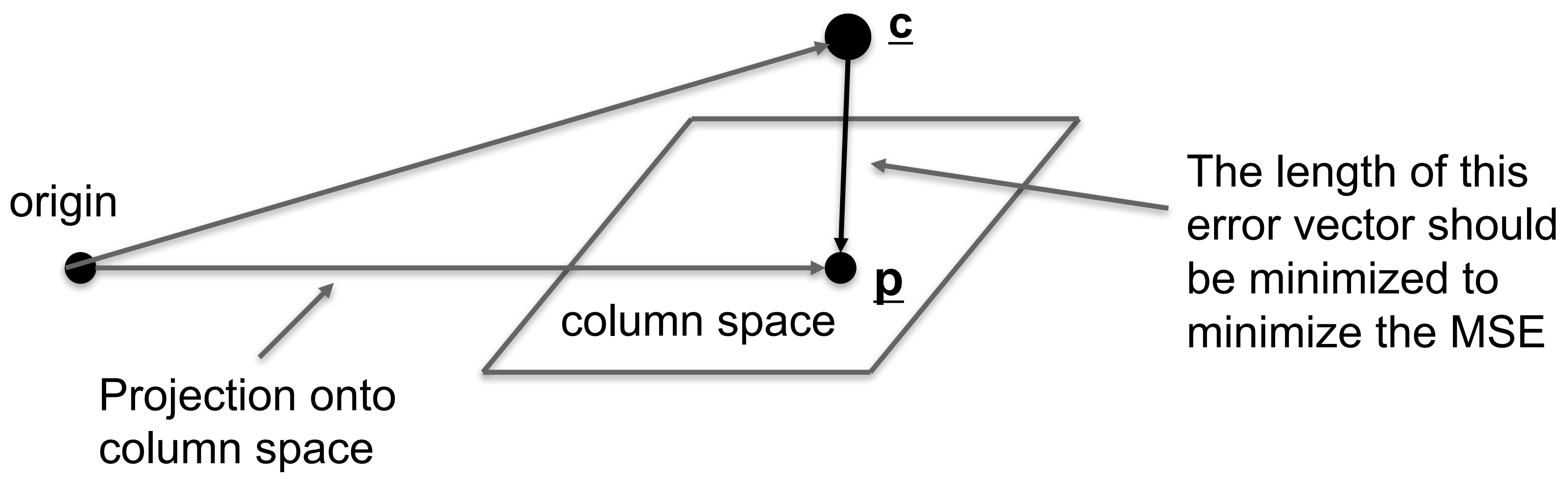
$$E = | A\underline{\mathbf{x}} - \underline{\mathbf{c}} |^2$$

- Ultimately we want a matrix, A^+, the pseudo inverse, such that the MSE solution, $\underline{\mathbf{x}}^{\$}$, is given by

$$\underline{\mathbf{x}}^{\$} = A^+ \underline{\mathbf{c}}$$

Pseudo Inverse cont.

- **What is the closest point in the column space of A to <u>c</u>?**
  - Answer: the *projection*, <u>p</u>, of <u>c</u> onto the column space
    - ✓ Why? Because moving *orthogonally* from <u>c</u> to a point in the column space is the shortest distance path
      - – Think about the 3-D case and moving from an arbitrary point to a 2-D plane. You should move orthogonally


c
origin
p
column space
Projection onto column space
The length of this error vector should be minimized to minimize the MSE

Pseudo Inverse cont.

- **What does this mean in terms of math?**
  - To be orthogonal to the column space, the error vector, $\underline{\mathbf{c}} - \underline{\mathbf{p}}$, must be orthogonal to *every* column of A
  - As an equation, this requirement becomes

$$(\underline{\mathbf{c}} - \underline{\mathbf{p}})^T A = \underline{\mathbf{0}}$$

$$\underline{\mathbf{p}}^T A = \underline{\mathbf{c}}^T A$$

$$A^T \underline{\mathbf{p}} = A^T \underline{\mathbf{c}} \longleftarrow$$

This equation says that both $\underline{\mathbf{c}}$ and the optimal point $\underline{\mathbf{p}}$ have the same projections onto the columns of A

Pseudo Inverse cont.

CU Boulder

- **Let $\underline{\mathbf{x}}^{\$}$ be the solution we seek (i.e. it is optimal in the sense of minimizing E)**
  - As an equation: A$\underline{\mathbf{x}}^{\$}$ = $\underline{\mathbf{p}}$
  - *Assume the columns of A are linearly independent* (i.e. only one combination of the columns will produce $\underline{\mathbf{p}}$). Substituting we obtain

$$A^T A\underline{\mathbf{x}}^{\$} = A^T \underline{\mathbf{c}}$$

$$\underline{\mathbf{x}}^{\$} = (A^T A)^{-1} A^T \underline{\mathbf{c}}$$

  - In this case, we see that the pseudo inverse of A is given by

$$A^+ = (A^T A)^{-1} A^T$$

Pseudo Inverse cont.

CU Boulder

- **If the columns of A are *not* linearly independent, then there are multiple solutions to A$\underline{x}$ = $\underline{p}$**
  - We must *define* what we mean by the optimal solution
  - Criterion: of all possible solutions, we choose the one with minimum length
- **The pseudo inverse matrix satisfying $\underline{x}^{\$}$ = $A^{+}\underline{c}$ can be found for this case using the Singular Value Decomposition (SVD)**