Things to do

1. Take the Quiz
2. Get your environment set up
    1. Open a terminal window
    2. create a class02 “mkdir class02”
    3. change directory to class02 “cd class02”
    4. start ipython

Working with Arrays and Matrices in Python

Class #2 – Phys 281

Grant Wilson

Why arrays and matrices?

- Any set of data can be packed into an array (or a multi-dimensional array which we'll call a matrix).
  - a set of time-ordered data (and the time values themselves!)
  - the x or y values of a series of related vectors
  - the titles of movies you've watched in Netflix
  - the ratings you gave those movies
- Once the data is organized in array form, you have all the tools of Linear Algebra to work with it.
- Arrays can also be plotted.
- Two dimensional images are simply two dimensional arrays.
- Note that almost all programming is "vector-based" now so this is really important stuff to learn.

Arrays in Computer Memory

- Arrays exist in the physical memory of the computer.
  - They have a size and a length.
  - They have a set of memory addresses
- Warning – two different variables can point at the same memory address(es). This can be a point of confusion.
- It's also possible to use up all the available memory … which is bad.

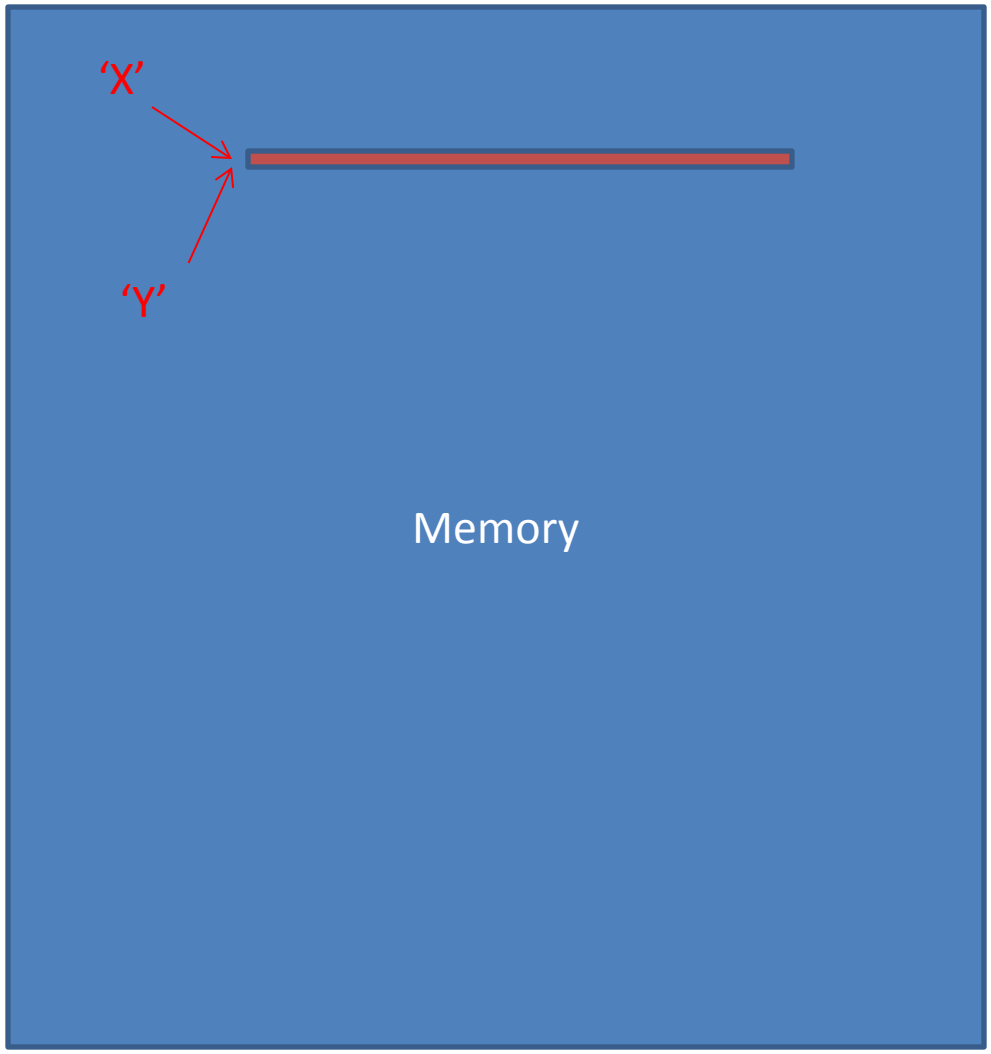
Suppose ‘X’ is a list of all
the people who died in Breaking Bad.

Beware:
If I then type “Y=X” then the variable ‘Y’ points to the same location in memory!

So “Y[0] = ‘Walt’” changes the value of X[0] too.

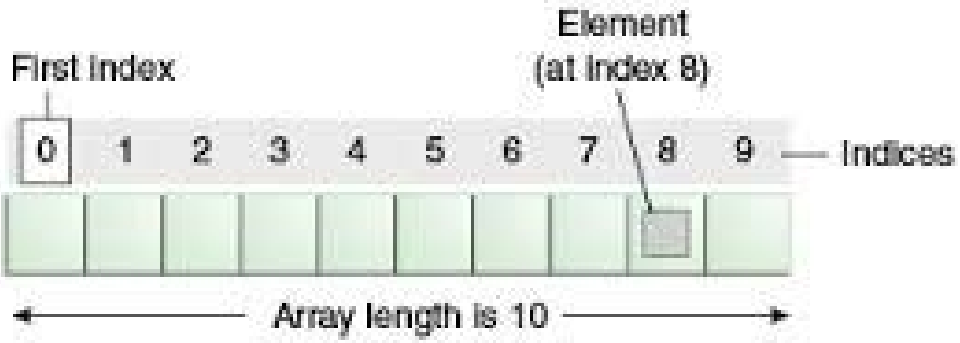
If you want to make a copy of X and call it Y, then you use:

```
Y = list(X)          #if it’s a list, or
Y = X.copy()         #if it’s an array
```



Array Indices

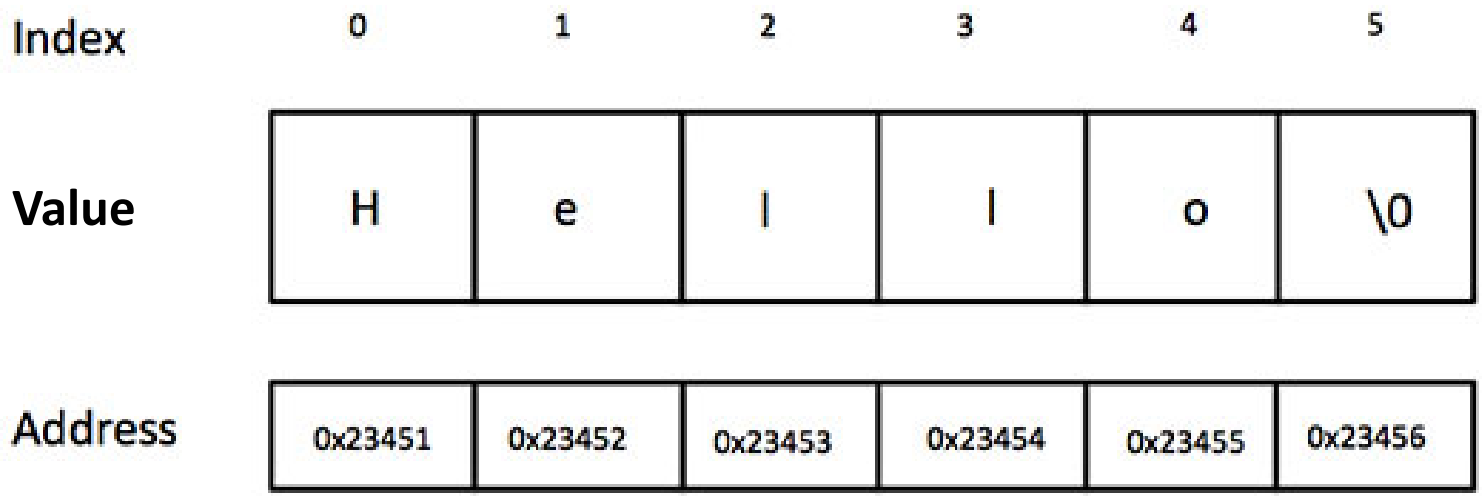
- Often it is convenient to look at particular members of an array via its index.



- Python indexing starts at zero (like your age in years and building floors in Europe)

- Arrays of indexes are extremely useful (though can also be very confusing at first – we will practice this).

So there are three aspects to think about



Lists, arrays, and matrices

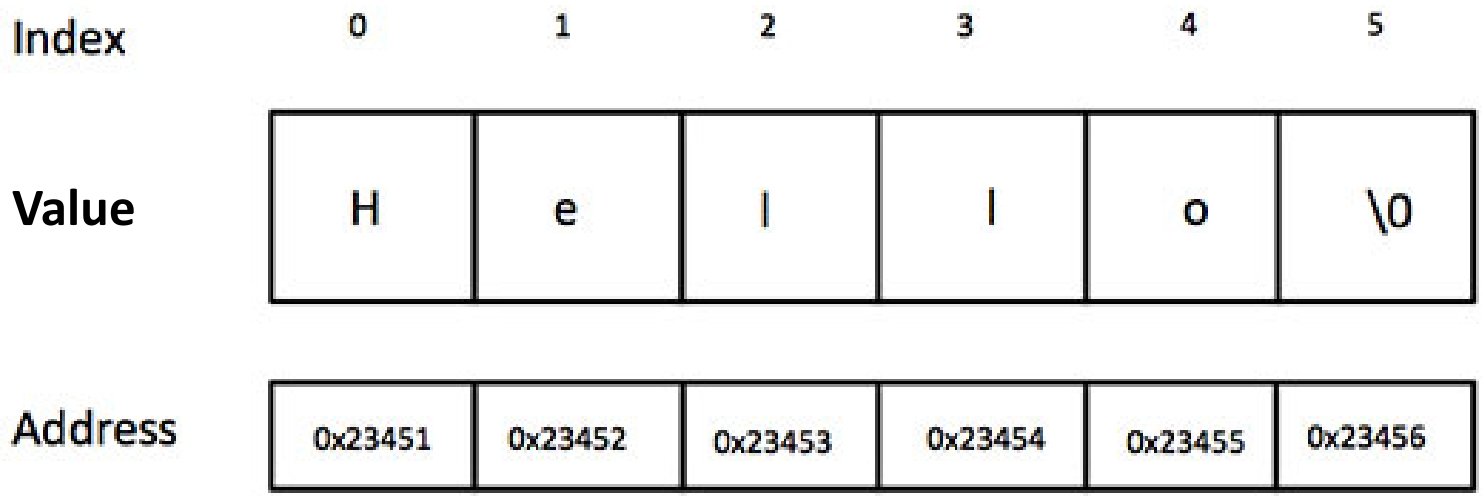
- Lists – these are lists of things (really anything)
- arrays – specifically, numpy arrays – these are mathematical containers of similar objects (see below)
- matrices – I wouldn't bother using these since they are just 2-dimensional numpy arrays.

Data Types

- Every value stored in a computer has a type
  - int – integer – … -2, -1, 0, 1, 2, …
  - float – floating point number – 1.5, 2.2, 1., 0.
  - string – “This is a string.”
  - complex – complex number – 4+5j

- Each type has a corresponding size (in memory) and so ints and floats are truly different beasts.

So there are three aspects to think about



Data Types

- Every value stored in a computer has a type
  - int – integer – … -2, -1, 0, 1, 2, …
  - float – floating point number – 1.5, 2.2, 1., 0.
  - string – “This is a string.”
  - complex – complex number – 4+5j

- Each type has a corresponding size (in memory) and so ints and floats are truly different beasts.

- Typecasting is the process of converting between types.
- Try this:
  - *type(1)*
  - *type(1.)*
  - *type(1+1.)*

Lists

- Don't overthink this, it's just a list.
  Like your grocery list, lists can
  contain different types.
  - *a = [0,0,4]*
  - *x = [1, 5.5, 'tree', 'apple']*

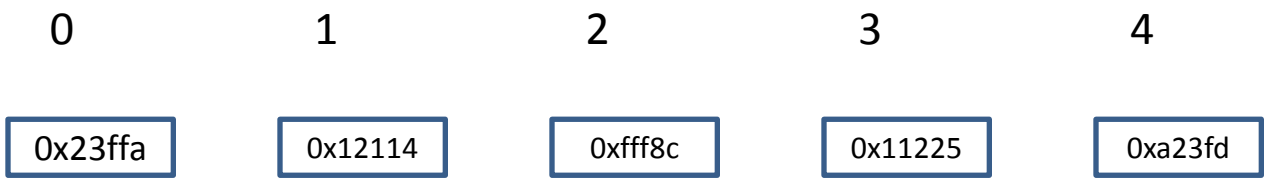
- Lists can be efficiently appended to:
  - *x.append('leaf')*
  - *print x*
  - *print x+a*

- And lists can be easily cycled through:
  - *for object in x:*
    print object

- But lists are NOT contiguous in physical memory

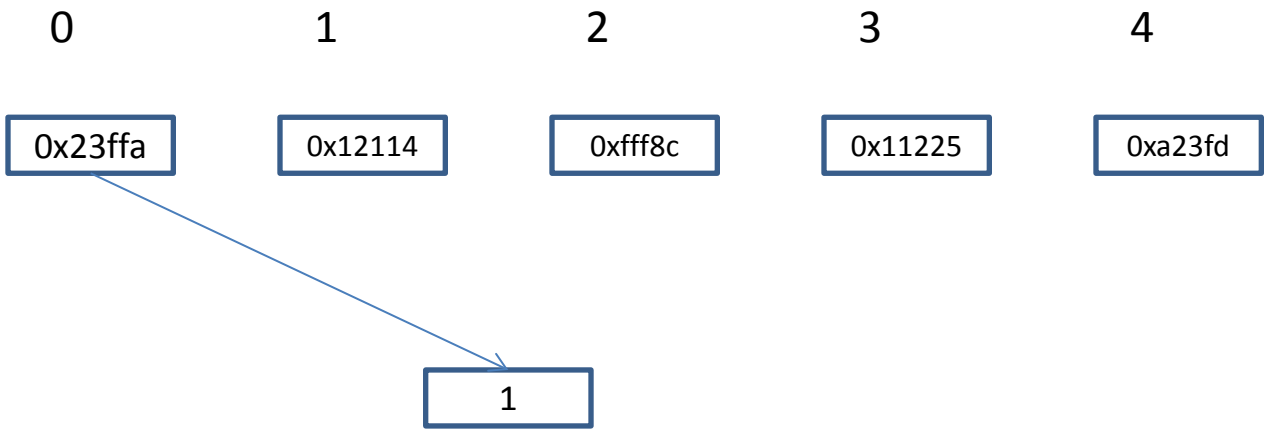
Lists are not contiguous in memory

- A python list is an array of memory addresses that point to the values in the list



Lists are not contiguous in memory

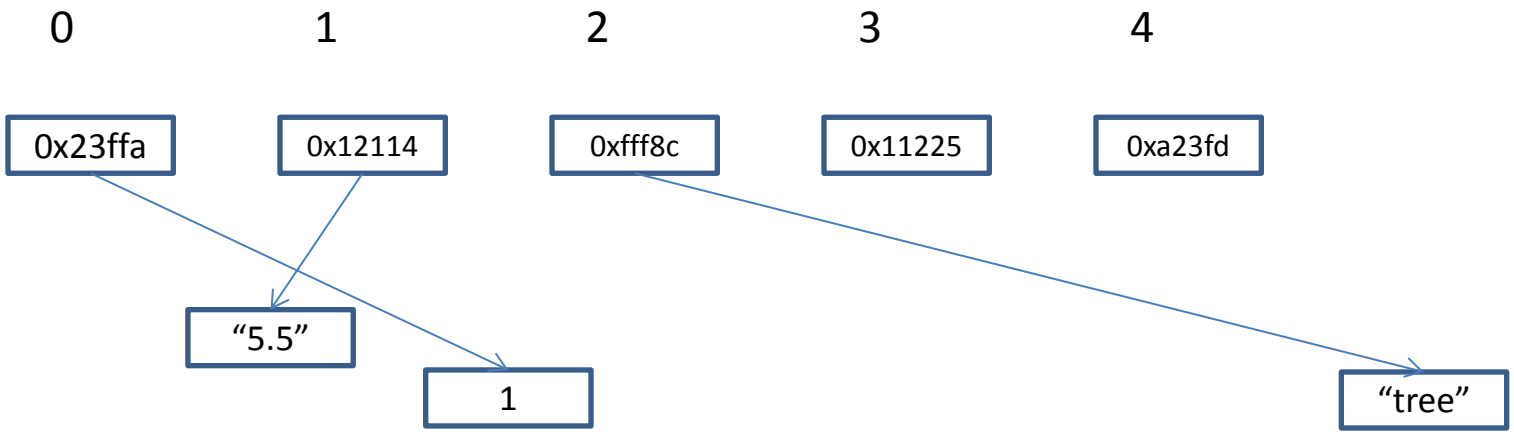
- A python list is an array of memory addresses that point to the values in the list



- This can be just fine, but this is too inefficient for most demanding math applications.

Lists are not contiguous in memory

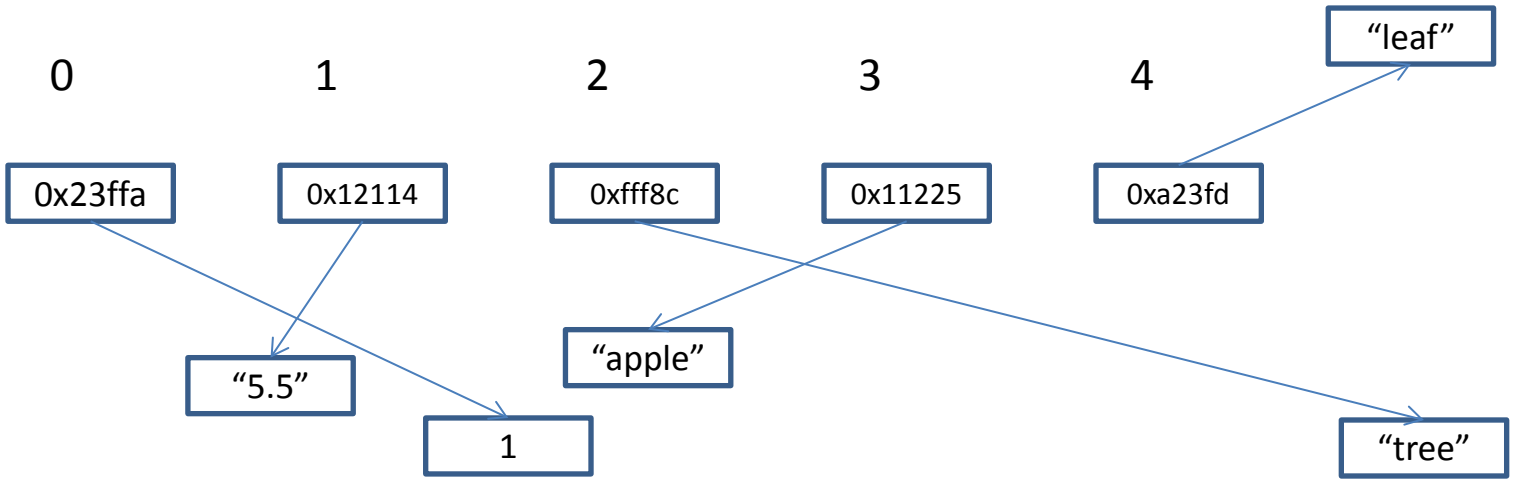
- A python list is an array of memory addresses that point to the values in the list



- This can be just fine, but this is too inefficient for most demanding math applications.

Lists are not contiguous in memory

- A python list is an array of memory addresses that point to the values in the list



- This can be just fine, but this is too inefficient for most demanding math applications.

Arrays

- Arrays are *containers* of a homogeneous collection of data (ie, fixed size and type)
  - arrays of integers
  - arrays of floats
  - arrays of complex numbers
  - etc.
- Arrays have a *shape* (dimensionality)
- Contents of an array can be accessed through their indices.

Arrays (cont.)

- We will exclusively use numpy arrays for all our math.
  - This gives us access to lots of ancillary information about our arrays.

- At the top of your program (or inside ipython) you must have
  - *import numpy as np*

Practice with Arrays

- Create a 1-d array of the integers between 1 and 8
  - *x = np.array([1,2,3,4,5,6,7,8])*
- Print the data type of x
  - *print x.dtype*
- Print the length of x
  - *print len(x)*
- Print the shape of x
  - *print x.shape*
- Print the 1st element of x
  - *print x[0]*
- Print the 6th element of x
  - *print x[5]*
- Now define another array and add it to x
  - y = np.array([3,4,5])
  - print x+y

Some handy array creation tools

- Make an array of floats from 1. to 10.
  - *x = np.arange(1,11,dtype=float)*
  - *x = np.arange(1.,11.)*
- Make an array of ints from 0 to 9
  - *x = np.arange(10)*
- Make an array of floats from 10. to 1.
  - *x = np.arange(10,0,-1,dtype=float)*
- Make a linearly spaced vector between 1 and 10 with seven points
  - *x = np.linspace(1,10,7)*
- *Make an N-element array filled with zeros.*
  - *x = np.array([0.]*N)*

Practice

- Create an array of all the years since your birth including your birth year and the present year.
- Create a list of all the people in your immediate family. Create a matching array with their ages.
- Find a one-line method of calculating the sum of the ages of everyone in your family using the array you created above.
- Write a script to have the computer identify the oldest and youngest members of your family.

Answer to the last practice problem

- Suppose my family list is
  - *family = [“Homer”, “Marge”, “Bart”, “Lisa”, “Maggie”]*
- And the corresponding ages are:
  - *ages = np.array([40, 37, 10, 8, 1])*
- To find the oldest in the ages array:
  - *oldest = ages.max()*
  - *print “Oldest person in my family is “, oldest, “ years old”*
- To find the corresponding index use:
  - *i_oldest = np.where(ages == oldest)*
- And the find the character with that index use:
  - *oldest_family_member = family[i_oldest[0]]*
  - *print “Their name is “, oldest_family_member*

Answer to the last practice problem

- Suppose my family list is
  - *family = [“Homer”, “Marge”, “Bart”, “Lisa”, “Maggie”]*
- And the corresponding ages are:
  - *ages = np.array([40, 37, 10, 8, 1])*
- To find the oldest in the ages array:
  - *oldest = ages.max()*
  - *print “Oldest person in my family is “, oldest, “ years old”*
- To find the corresponding index use:
  - *i_oldest = np.where(ages == oldest)*
- And the find the character with that index use:
  - *oldest_family_member = family[i_oldest[0]]*
  - *print “Their name is “, oldest_family_member*

Answer to the last practice problem

- Suppose my family list is
  - *family = [“Homer”, “Marge”, “Bart”, “Lisa”, “Maggie”]*
- And the corresponding ages are:
  - *ages = np.array([40, 37, 10, 8, 1])*
- To find the oldest in the ages array:
  - *oldest = ages.max()*
  - *print “Oldest person in my family is “, oldest, “ years old”*
- To find the corresponding index use:
  - *i_oldest = np.where(ages == oldest)*
- And the find the character with that index use:
  - *oldest_family_member = family[i_oldest[0]]*
  - *print “Their name is “, oldest_family_member*

Python Array Math Doesn't Always Seem Intuitive

- Add 7 to all elements of an array
  - *x = np.array([1.,2.,3.])+7.*
- Multiply all array elements by 4.
  - *x = np.arange(10)*4.*
- *Multipy two arrays together, element by element*
  - *x = np.array([2.,2.,4.,4.])*
  - *y = np.array([0.,1.,2.,3.])*
  - *z = x*y*
- Take the dot-product of two arrays
  - *x = np.arange(3.)*
  - *y = np.array([3.,4.,1.])*
  - *z = np.dot(x,y) or*
  - *z = x.dot(y)*

Looping through elements of an array/list (one of the most useful things to know)

- Do it directly with for/in
  - *x = np.linspace(1,8,15)*
  - *for xi in x:*
    print xi

- Use enumerate to keep track of the index in an array/list
  - *x = np.linspace(1,8,15)*
  - *y = x+8.*
  - *for i, xi in enumerate(x):*
    print i, xi, x[i], y[i]

- Or just do it manually using range()
  - *x = np.linspace(1,8,15)*
  - *for i in range(len(x)):*
    print i, x[i]

The ":" operator and other indexing tricks

- Using : is called "slicing" the array
  - *x = np.arange(0:10)*
  - *print x*
  - *print x[:]*
  - *print x[0:4]*
  - *print x[:4]*
  - *print x[5:]*

- *Note that negative indices wrap to the end of the array*
  - *print x[-1]*

  (this is very dangerous!)

An aside on being cute when programming

- Here's a piece of c code that can correctly give the political affiliations of the past 31 US presidents (by Adrian Cable).
  - *main(int riguing,char**acters){puts(1[acters-~!(*(int*)1[acters]%4796%275%riguing)]);}*
- Here's another interesting one – this time in Python:
  - (lambda _, __, ___, ____, _____, ______, _______, ________, _________: getattr(__import__(True.__class__.__name__[_] + [].__class__.__name__[__]), ().__class__.__eq__.__class__.__name__[:__] + ().__iter__().__class__.__name__[_____:_________])(_, (lambda _, __, ___: _(_, __, ___))(lambda _, __, ___: chr(___ % __) + _(_, __, ___ // __) if ___ else (lambda: _).func_code.co_lnotab, _ << _________, (((_____ << ____) + _) << ((___ << _____) - ___)) + (((((___ << __) - _) << ___) + _) << ((_____ << ____) + (_ << _))) + (((________ << __) - _) << ((((_ << ___) + _)) << ___) + (_ << _))) + (((________ << ___) + _) << ((_ << ______) + _)) + (((________ << ____) - _) << ((________ << ___))) + (((_ << _____) - _) << ((((___ << __) + _) << __) - _)) - (________ << ((((___ << __) - _) << __) + _)) + (________ << (((((_ << ___) + _)) << __))) - ((((((_ << ___) + _)) << __) + _) << (((___ << __) + _) << _))) + (((________ << __) - _) << (((((_ << ___) + _)) << _))) + (((___ << ___) + _) << ((_____ << _))) + (_____ << ______) + (_ << ___))))(*(lambda _, __, ___: _(_, __, ___))((lambda _, __, ___: [__(___[(lambda: _).func_code.co_nlocals])] + _(_, __, ___[(lambda _: _).func_code.co_nlocals:]) if ___ else []), lambda _: _.func_code.co_argcount, (lambda _: _, lambda _, __: _, lambda _, __, ___: _, lambda _, __, ___, ____: _, lambda _, __, ___, ____, _____: _, lambda _, __, ___, ____, _____, ______: _, lambda _, __, ___, ____, _____, ______, _______: _, lambda _, __, ___, ____, _____, ______, _______, ________: _)))

An aside on being cute when programming

- Readability will serve you far better in the long run than cuteness.
  - Avoid: *y = x[-2]*
  - Instead: *y = x[len(x)-2]*

  - *Avoid: a,b,c = cos(x), sin(x), tan(x)*

- Especially avoid cuteness when developing a new piece of code!