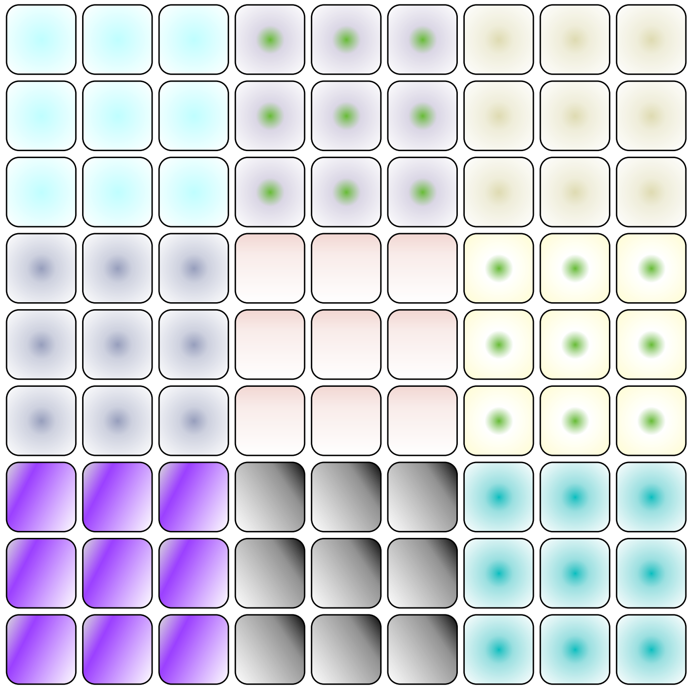
Sudoku Problem

Thomas Schwarz, SJ

Sudoku

- $81 = 9 \times 9$ squares
  - Filled with numbers from 1 to 9
  - All numbers from 1 to 9 need to make up:
    - Each of nine rows
    - Each of nine columns
    - Each of nine "houses"
      - A 3×3 sub-block



Sudoku

- Task:
  - Write a valid Sudoku test for a 9×9 integer array in numpy

Checking for Equality

- We want to check whether a row has exactly the same elements as np.arange(1,10)

Checking for Equality

- To check whether two arrays are equal
  - Use np.array_equal to check whether two arrays are equal
  - Checks whether shape is the same and whether all elements in the same position are equal
    - `np.array_equal(np.array([1,2,3,4]), np.array([3,2,1,4]))` is False
    - Because they are different as array but not as sets

Checking for Equality

- This type of equality is of arrays,
    - but not of sets
- To check whether one array is a permutation of another:
    - Use numpy operations
    - Build something ourselves

Checking for Equality

- Can use set operations in numpy
  - `in1d(array1, array2)` tests whether each element in the first array is also in the second
  - `intersect1d(array1, array2)`: intersection of the arrays
  - `setdiff1d(array1, array2)`: find the set difference
    - Those in array1 that are not in array2
  - `setxor1d(array1, array2)`: symmetric difference of the arrays
  - `union1d(array1, array2)`: union, not concatenation of the two arrays

Checking for Equality

- `isin(element, array)`:
  - Checks whether element is in the array
  - This is useful
  - But be careful about broadcasting and flattening

```
test=np.array([
   [1,2,4,5,3],
   [4,4,3,1,5],
   [1,4,3,2,5],
   [1,2,5,0,2],
   [1,1,2,3,5]
   ])
```

  - `np.isin(np.arange(1,6), test)`
    - just returns [True, True, True, True, True] as the test array gets flattened and the function is applied on the first parameter

Checking for Equality

- `isin(element, array):`
    - We can combine the results of isin by using np.all or np.any
        - np.all(array) returns True if all elements of the array are true
        - np.any(array) returns True if at least one element of the array is true

Checking for Equality

- We now can test whether an array of nine numbers is a permutation of {1,2,3,4,5,6,7,8,9}
    - Assume that we know beforehand that the array has nine members
    - Then if all of 1, 2, 3, …, 9 are in the array, the array is a permuation

Checking for Equality

- We need to combine
    - Cardinality
    - IsIn
    - All

```
def check_perm(array):
    return np.all(np.isin(np.arange(1,10), array)) and
len(array)==9
```

Numpy Array Indexing Repetition

- Array indexing is vital, but the many possibilities in numpy are confusing
- So, let's look at them again

Numpy Indexing

- First, generate a simple array made into a 3×3 matrix

```
>>> mat = np.arange(1,10)
>>> mat = np.reshape(mat,(3,3))
>>> print(mat)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Numpy Indexing

- The simplest indexing is direct, python-style

```
>>> mat[0,2]
3
```

- We can also address with tuples
  - In fact, `mat[0,2]` is internally converted into `mat.__getitem__((0,2))`

```
>>> mat[(0,2)]
3
```

Numpy Indexing

- Tuple addressing is in fact more "natural"

```
>>> for i in np.ndindex(mat.shape):
      print (i, mat[i])
```

```
(0, 0) 1
(0, 1) 2
(0, 2) 3
(1, 0) 4
(1, 1) 5
(1, 2) 6
(2, 0) 7
(2, 1) 8
(2, 2) 9
```

Numpy Indexing

- Remember `zip(*iterable)` to aggregate a tuple?
- We can use np.where with a single argument, a condition, to find the elements in an array that satisfy the condition

```
>>> list(zip(*np.where(mat%2==0)))
[(0, 1), (1, 0), (1, 2), (2, 1)]
```

- np.where returns a list of indices where the condition is true
- We then convert the list into tuples

Numpy Slicing

- Normal slicing works as in Python
  - With the important difference
    - Slices are not copies
    - If you need copies, then use the copy( ) method

```
>>> A = mat.copy()
>>> print(A)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Numpy Slicing

- Slice actually constructs slice objects
  - You can apply them to all arrays (of the correct shape)

```
>>> s = slice(1,None, 2)
>>> print(s)
slice(1, None, 2)
>>> arr = np.arange(1,10)
>>> arr
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> arr[s]
array([2, 4, 6, 8])
```

Numpy Slicing

- Slices have up to three components:
  - Start (default 0 or -1)
  - Stop (default 0 or -1)
  - Step (default 1)
- For multidimensional arrays, can slice in each dimension
  - First part is rows
  - Second part is columns
  - etc

Numpy Slicing

- Examples:
    - Inverting rows: use first dimension

```
>>> A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> A[::-1,:]
array([[7, 8, 9],
       [4, 5, 6],
       [1, 2, 3]])
```

Numpy Slicing

- Examples:
  - Invert columns: Use second dimension

```
>>> A[:,::-1]
array([[3, 2, 1],
       [6, 5, 4],
       [9, 8, 7]])
```

Numpy Slicing

- Examples: Get the first three rows of the first two columns

```
>>> B = np.arange(1,21)
>>> B = np.reshape(B,(4,5))
>>> print(B)
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
>>> B[0:3,0:2]
array([[ 1,  2],
       [ 6,  7],
       [11, 12]])
```

Numpy Slicing

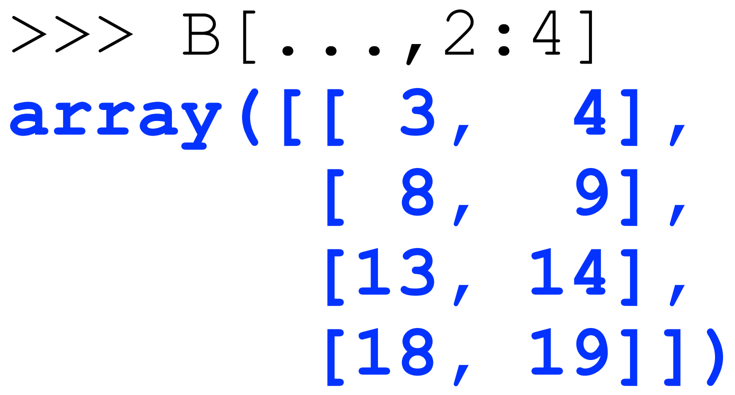
- If you specify a slice that has only one element, the dimension **does not** vanish

```
>>> B[1:2,3:4]
array([[9]])
```

  - The slice is still a two-dimensional array
    - Count the brackets
      - It just happens to have one row and one column

Numpy Slicing

- Ellipsis
  - As a short-cut, we can use the ellipsis, consisting of three dots
    - ...
    - Consists of as many colons as needed : :
      - But usually, we can only use one to avoid ambiguity
  -



```
>>> B[...,2:4]
array([[ 3,  4],
       [ 8,  9],
       [13, 14],
       [18, 19]])
```

Numpy Slicing

- If you do not provide enough information for each dimension, an ellipsis will be provided

Fancy Indexing

- Sometimes, slices are not enough
  - Then we can use fancy indexing
  - Example: Create a random two-dimensional array

```
>>> X = 10*numpy.random.rand(5,4)-5
>>> X
array([[ 1.23489451,  1.22443527,  3.35876328,  2.72987117],
       [-1.32420494,  4.14354623, -3.09531196,  4.97524407],
       [-4.43644932,  4.7533215 , -1.14004859, -4.32039428],
       [ 2.05397116, -1.05290493, -3.20528586, -4.5549263 ],
       [ 3.62748115,  0.53619237,  2.48564965,  1.34442926]])
```

Fancy Indexing

- Let's square the negative entries
- Need to change X

```
>>> X[X<0] **= 2
>>> X
array([[ 1.23489451,  1.22443527,  3.35876328,  2.72987117],
       [ 1.75351872,  4.14354623,  9.58095613,  4.97524407],
       [19.68208255,  4.7533215 ,  1.29971079, 18.66580675],
       [ 2.05397116,  1.1086088 , 10.27385742, 20.74735363],
       [ 3.62748115,  0.53619237,  2.48564965,  1.34442926]])
```

Numpy Slicing

- You can always use slices to assign to numpy arrays
- This is different then for Python

- QUIZ:
  - Create an array from 1 to 20.
  - Change this to a 5 by 4 array
  - Change the second row to negatives

Numpy Slicing

- Answer:

```
>>> import numpy as np
>>> x = np.reshape(np.arange(1,21),(5,4))
>>> x
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20]])
>>> x[1]
array([5, 6, 7, 8])
>>> x[1,...]=-x[1,...]
>>> x
array([[ 1,  2,  3,  4],
       [-5, -6, -7, -8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20]])
```

Numpy Slicing

- QUIZ:
  - No change the second row by squaring

```
>>> x[...,1]=x[...,1]**2
>>> x
array([[  1,   4,   3,   4],
       [ -5,  36,  -7,  -8],
       [  9, 100,  11,  12],
       [ 13, 196,  15,  16],
       [ 17, 324,  19,  20]])
```

Sudoku Task

- Now we have all the ingredients for you to write a program that checks the Sudoku:
  - You can check whether an array is a permutation of 1 … 9
  - You can extract rows
  - You can extract columns
  - You can extract houses (sub-squares)
  - And you can combine checks with np.all