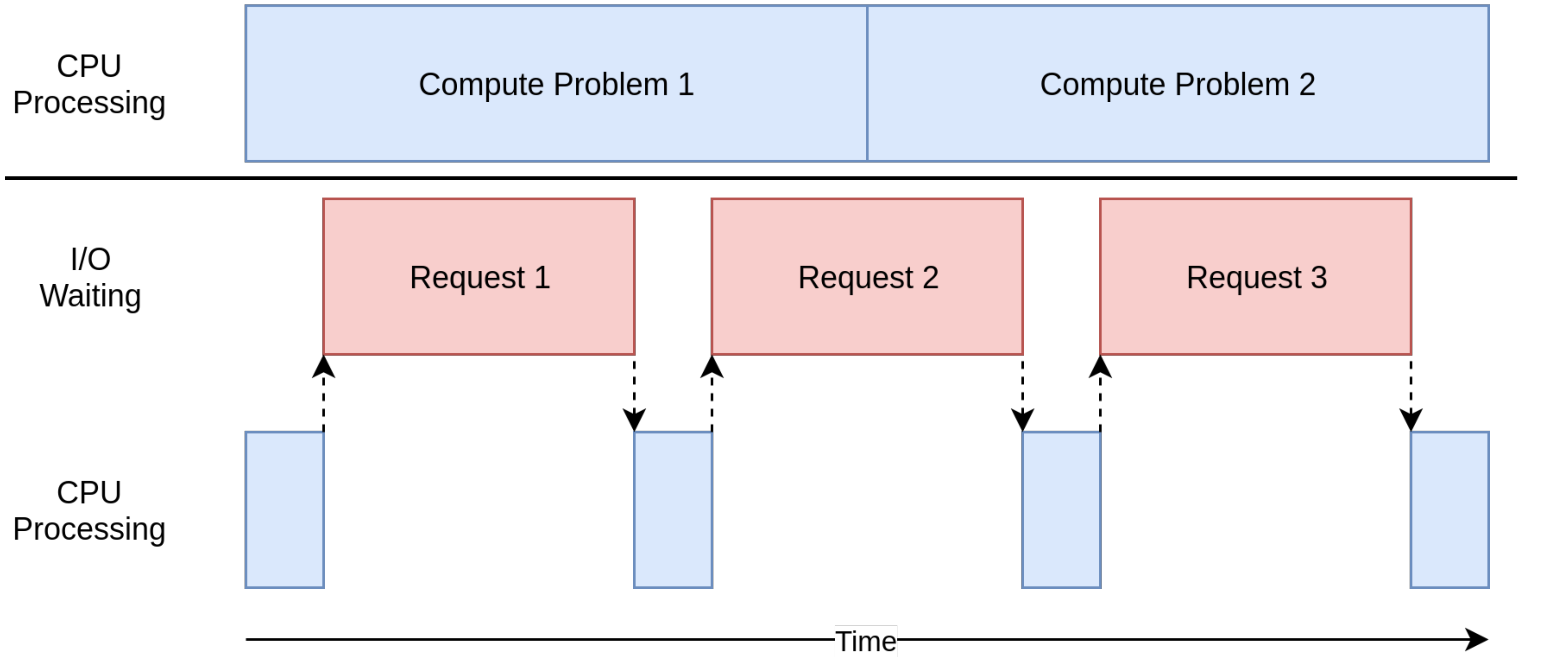


Programming Principles in Python (CSCI 503/490)

Arrays

Dr. David Koop

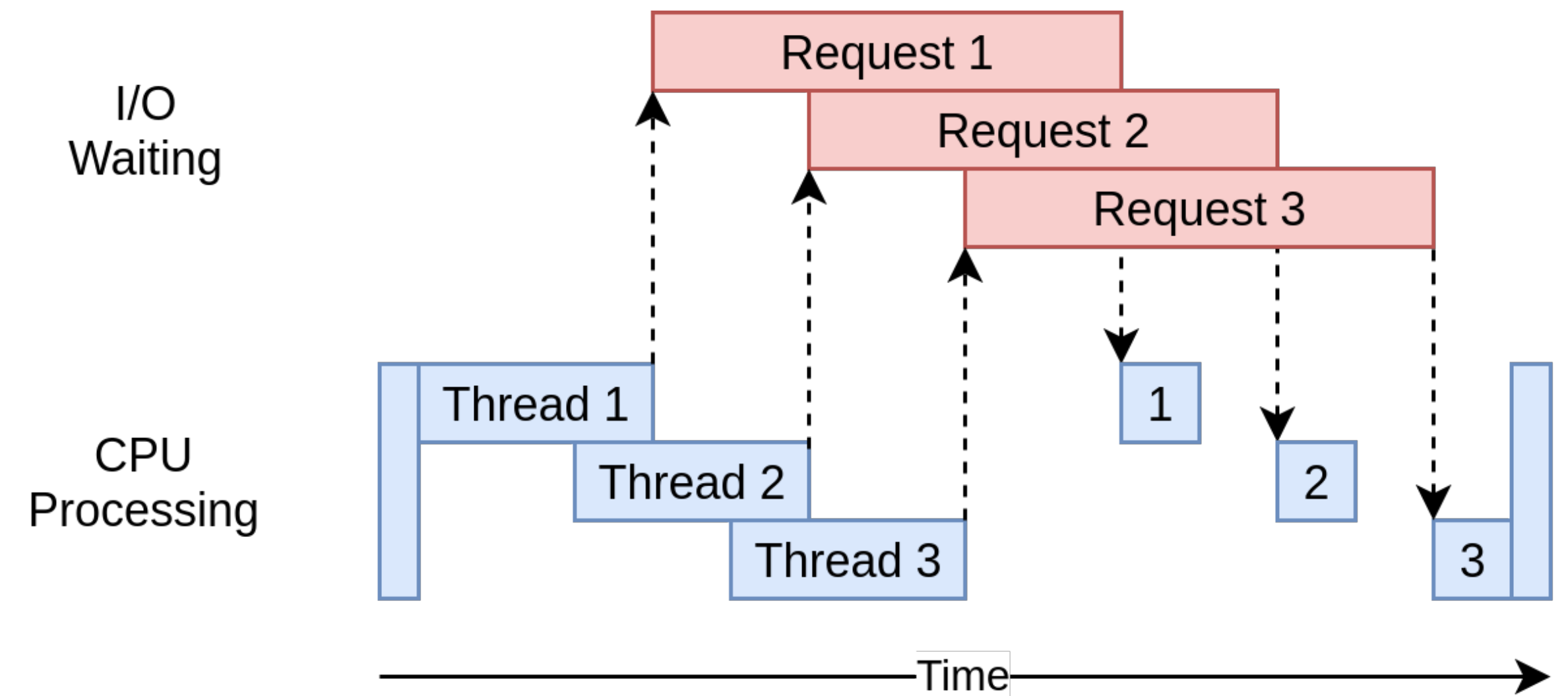
CPU-Bound vs. I/O-Bound



[J. Anderson]

Threading

- Threading address the I/O waits by letting separate pieces of a program run at the same time
- Threads run in the same process
- Threads share the same memory (and global variables)
- Operating system schedules threads; it can manage when each thread runs, e.g. round-robin scheduling
- When blocking for I/O, other threads can run



[J. Anderson]

Python Threading Speed

- If I/O bound, threads work great because time spent waiting can now be used by other threads
- Threads **do not** run simultaneously in standard Python, i.e. they cannot take advantage of multiple cores
- Use threads when code is **I/O bound**, otherwise no real speed-up plus some overhead for using threads

Python and the GIL

- Solution for reference counting (used for garbage collection)
- Could add locking to every value/data structure, but with multiple locks comes possible **deadlock**
- Python instead has a Global Interpreter Lock (GIL) that must be acquired to execute any Python code
- This effectively makes Python single-threaded (faster execution)
- Python requires threads to give up GIL after certain amount of time
- Python 3 improved allocation of GIL to threads by not allowing a single CPU-bound thread to hog it

Multiprocessing

- Multiple processes do not need to share the same memory, interact less
- Python makes the difference between processes and threads minimal in most cases
- Big win: can take advantage of multiple cores!

Multiprocessing using concurrent.futures

- ```
import concurrent.futures
import multiprocessing as mp
import time

def dummy(num):
 time.sleep(5)
 return num ** 2

with concurrent.futures.ProcessPoolExecutor(max_workers=5,
 mp_context=mp.get_context('fork')) as executor:
 results = executor.map(dummy, range(10))
```
- `mp.get_context('fork')` changes from `'spawn'` used by default in MacOS, works in notebook

# asyncio

---

- Single event loop that controls when each task is run
- Tasks can be ready or waiting
- Tasks are **not interrupted** like they are with threading
  - Task controls when control goes back to the main event loop
  - Either waiting or complete
- Event loop keeps track of whether tasks are ready or waiting
  - Re-checks to see if new tasks are now ready
  - Picks the task that has been waiting the longest
- `async` and `await` keywords
- Requires support from libraries (e.g. `aiohttp`)

[J. Anderson]

# When to use threading, asyncio, or multiprocessing?

---

- If your code has a lot of I/O or Network usage:
  - If there is library support, use asyncio
  - Otherwise, multithreading is your best bet (lower overhead)
- If you have a GUI
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU bound:
  - You should use multiprocessing (if your machine has multiple cores)

[J. Anderson]

# Concurrency Comparison

| Concurrency Type                     | Switching Decision                                                    | Number of Processors |
|--------------------------------------|-----------------------------------------------------------------------|----------------------|
| Pre-emptive multitasking (threading) | The operating system decides when to switch tasks external to Python. | 1                    |
| Cooperative multitasking (asyncio)   | The tasks decide when to give up control.                             | 1                    |
| Multiprocessing (multiprocessing)    | The processes all run at the same time on different processors.       | Many                 |

[J. Anderson]

# Assignment 6

---

- Object-Oriented Programming & Exceptions
- Classes for an online market
- Use inheritance
- Due today

# Assignment 7

---

- Soon...
- Concurrency, Arrays

# Arrays

---

What is the difference between an array and a list (or a tuple)?

# Arrays

---

- Usually a fixed size—lists are meant to change size
- Are mutable—tuples are not
- Store only one type of data—lists and tuples can store any combination
- Are faster to access and manipulate than lists or tuples
- Can be multidimensional:
  - Can have list of lists or tuple of tuples but no guarantee on shape
  - Multidimensional arrays are rectangles, cubes, etc.

# Why NumPy?

---

- Fast **vectorized** array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application).

[W. McKinney, Python for Data Analysis]

```
import numpy as np
```

# Creating arrays

---

- `data1 = [6, 7, 8, 0, 1]`  
`arr1 = np.array(data1)`
- `data2 = [[1.5, 2, 3, 4], [5, 6, 7, 8]]`  
`arr2 = np.array(data2)`
- `data3 = np.array([6, "abc", 3.57])` # !!! check !!!
- Can check the type of an array in `dtype` property
- Types:
  - `arr1.dtype` # `dtype('int64')`
  - `arr3.dtype` # `dtype('<U21')`, unicode plus # chars

# Types

---

- "But I thought Python wasn't stingy about types..."
- numpy aims for speed
- Able to do array arithmetic
- int16, int32, int64, float32, float64, bool, object
- Can specify type explicitly
  - `arr1_float = np.array(data1, dtype='float64')`
- `astype` method allows you to convert between different types of arrays:

```
arr = np.array([1, 2, 3, 4, 5])
arr.dtype
float_arr = arr.astype(np.float64)
```

# numpy data types (dtypes)

| Type                                    | Type code       | Description                                                                                                            |
|-----------------------------------------|-----------------|------------------------------------------------------------------------------------------------------------------------|
| int8, uint8                             | i1, u1          | Signed and unsigned 8-bit (1 byte) integer types                                                                       |
| int16, uint16                           | i2, u2          | Signed and unsigned 16-bit integer types                                                                               |
| int32, uint32                           | i4, u4          | Signed and unsigned 32-bit integer types                                                                               |
| int64, uint64                           | i8, u8          | Signed and unsigned 64-bit integer types                                                                               |
| float16                                 | f2              | Half-precision floating point                                                                                          |
| float32                                 | f4 or f         | Standard single-precision floating point; compatible with C float                                                      |
| float64                                 | f8 or d         | Standard double-precision floating point; compatible with C double and Python float object                             |
| float128                                | f16 or g        | Extended-precision floating point                                                                                      |
| complex64,<br>complex128,<br>complex256 | c8, c16,<br>c32 | Complex numbers represented by two 32, 64, or 128 floats, respectively                                                 |
| bool                                    | ?               | Boolean type storing True and False values                                                                             |
| object                                  | O               | Python object type; a value can be any Python object                                                                   |
| string_                                 | S               | Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10' |
| unicode_                                | U               | Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')   |

[W. McKinney, Python for Data Analysis]

# Array Shape

---

- Our normal way of checking the size of a collection is... `len`
- How does this work for arrays?
- `arr1 = np.array([1, 2, 3, 6, 9])`  
`len(arr1) # 5`
- `arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])`  
`len(arr2) # 2`
- All dimension lengths → shape: `arr2.shape # (2, 4)`
- Number of dimensions: `arr2.ndim # 2`
- Can also reshape an array:
  - `arr2.reshape(4, 2)`
  - `arr2.reshape(-1, 2) # what happens here?`

# Speed Benefits

---

- Compare random number generation in pure Python versus numpy

- Python:

```
- import random
 %timeit rolls_list = [random.randrange(1,7)
 for i in range(0, 60_000)]
```

- With NumPy:

```
- %timeit rolls_array = np.random.randint(1, 7, 60_000)
```

- Significant speedup (80x+)

# Array Programming

---

- Lists:

- ```
c = []  
  for i in range(len(a)):  
      c.append(a[i] + b[i])
```

- How to improve this?

Array Programming

- Lists:

- `c = []`
 `for i in range(len(a)):`
 `c.append(a[i] + b[i])`
- `c = [aa + bb for aa, bb in zip(a,b)]`

- NumPy arrays:

- `c = a + b`

- More functional-style than imperative

- **Internal iteration** instead of external

Operations

- `a = np.array([1, 2, 3])`
`b = np.array([6, 4, 3])`
- (Array, Array) Operations (**Element-wise**)
 - Addition, Subtraction, Multiplication
 - `a + b` # `array([7, 6, 6])`
- (Scalar, Array) Operations (**Broadcasting**):
 - Addition, Subtraction, Multiplication, Division, Exponentiation
 - `a ** 2` # `array([1, 4, 9])`
 - `b + 3` # `array([9, 7, 6])`

More on Array Creation

- Zeros: `np.zeros(10)`
- Ones: `np.ones((4,5))` # shape
- Empty: `np.empty((2,2))`
- _like versions: pass an existing array and matches shape with specified contents
- Range: `np.arange(15)` # constructs an array, not iterator!

Indexing

- Same as with lists plus shorthand for 2D+
 - `arr1 = np.array([6, 7, 8, 0, 1])`
 - `arr1[1]`
 - `arr1[-1]`
- What about two dimensions?
 - `arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])`
 - `arr[1][1]`
 - `arr[1,1]` # shorthand

2D Indexing

		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

[W. McKinney, Python for Data Analysis]

Slicing

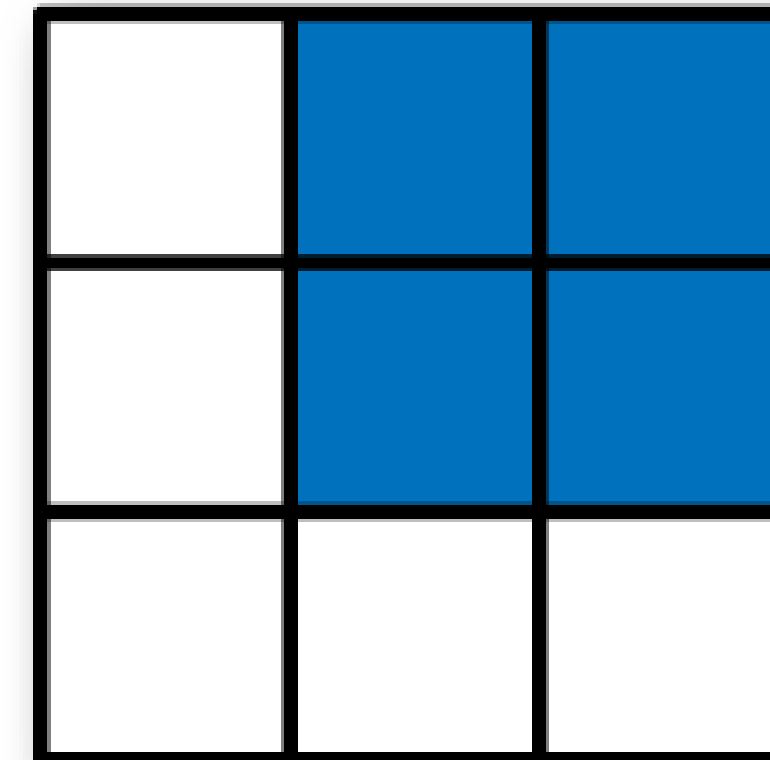
- 1D: Similar to lists
 - `arr1 = np.array([6, 7, 8, 0, 1])`
 - `arr1[2:5]` # `np.array([8, 0, 1])`, sort of
- Can **mutate** original array:
 - `arr1[2:5] = 3` # supports assignment
 - `arr1` # the original array changed
- Slicing returns **views** (copy the array if original array shouldn't change)
 - `arr1[2:5]` # a view
 - `arr1[2:5].copy()` # a new array

Slicing

- 2D+: comma separated indices as shorthand:
 - `arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])`
 - `a[1:3, 1:3]`
 - `a[1:3, :]` # works like in single-dimensional lists
- Can combine index and slice in different dimensions
 - `a[1, :]` # gives a row
 - `a[:, 1]` # gives a column

2D Array Slicing

How to obtain the blue slice
from array `arr`?



[W. McKinney, Python for Data Analysis]