

# Programming Principles in Python (CSCI 503/490)

---

## Arrays

Dr. David Koop

# Match Statement

---

- Python 3.10 added a match statement that can be used like a switch statement
- ```
match val:  
    case 1:  
        print('1st')  
    case 2:  
        print('2nd')  
    case _:  
        print('???')
```
- ... but this isn't better than if/elif or a dictionary dispatch
- The reason it was introduced is that it can do **more** than a switch statement

# Structural Pattern Matching

---

- Besides literal cases, match statements can be used to
  - differentiate structure
  - assign values
  - differentiate class instances
- Example:
- `match sys.argv:`
  - `case [_, "commit"]:`
    - `print("Committing")`
  - `case [_, 'add', fname]:`
    - `print("Adding file", fname)`

# Patterns

---

- Sequence Pattern:

```
match sys.argv:
    case [_, "commit"]:
        print("Committing")
    case [_, 'add', *fnames]:
        print("Adding files", fnames)
```

- Or and As Pattern:

```
match command.split():
    case ["go", ("north" | "south" | "east" | "west") as d]:
        current_room = current_room.neighbor(d)
```

# Mapping Pattern

---

- ```
for action in actions:  
    match action:  
        case {"text": message, "color": c}:  
            ui.set_text_color(c)  
            ui.display(message)  
        case {"sleep": duration}:  
            ui.wait(duration)  
        case {"sound": str(url), "format": "mp3"}:  
            ui.play(url)  
        case {"sound": _, "format": fmt, **rest}:  
            warning("Unsupported audio format", fmt, rest)
```
- Remember: Any unmatched key-value pairs are **ignored!**
- Can capture other pairs using `**rest`

# Class Pattern

---

- `@dataclass`  
`class Click:`  
    `x: float`  
    `y: float`  
    `button: Button # enum(LEFT, MIDDLE, RIGHT)`  
  
`for event in events:`  
    `match event:`  
        `case Click(x, y, button=Button.LEFT):`  
            `print("GOT a left click", x, y)`  
        `case Click():`  
            `print("GOT a click")`  
        `case _:`  
            `print("NO click")`

# Assignment 7

---

- Concurrency, System Integration, and Structural Pattern Matching
- Download System Logs
- Locate Logs of Interest
- Read JSON & Binary Data
- Filter suspicious events using structural pattern matching
- Process all files using threading

# 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, complex128, complex256	c8, c16, 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 aa, bb in zip(a, b):`  
        `c.append(aa + bb)`

- How to improve this?

# Array Programming

---

- Lists:

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

---

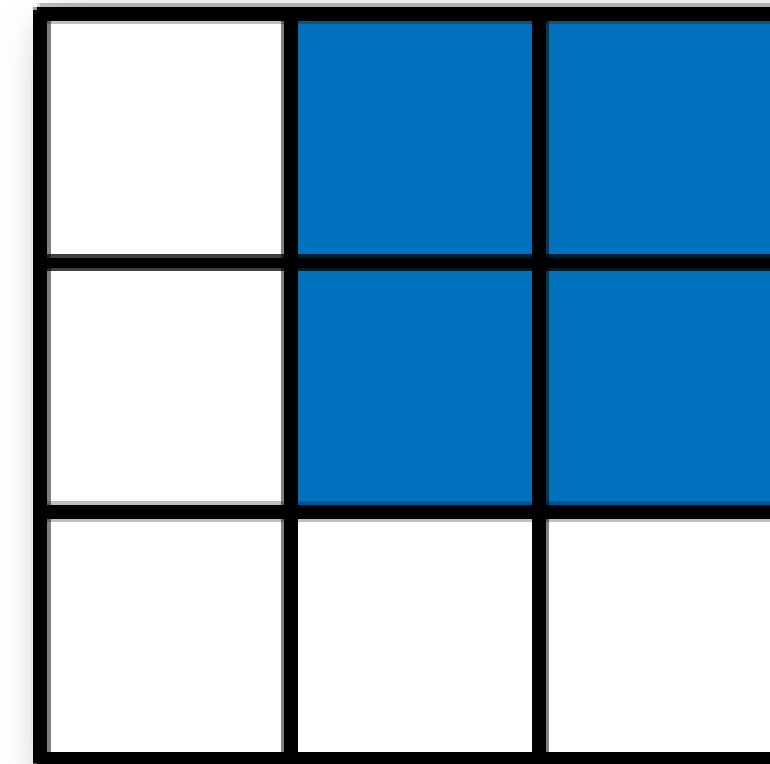
- 2D+: comma separated indices as shorthand:
  - `arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])`
  - `a[1:3, 1:3]`
  - `a[1:3, :]` # works like in single-dimensional lists
- Can combine index and slice in different dimensions
  - `a[1, :]` # gives a row
  - `a[:, 1]` # gives a column



# 2D Array Slicing

---

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

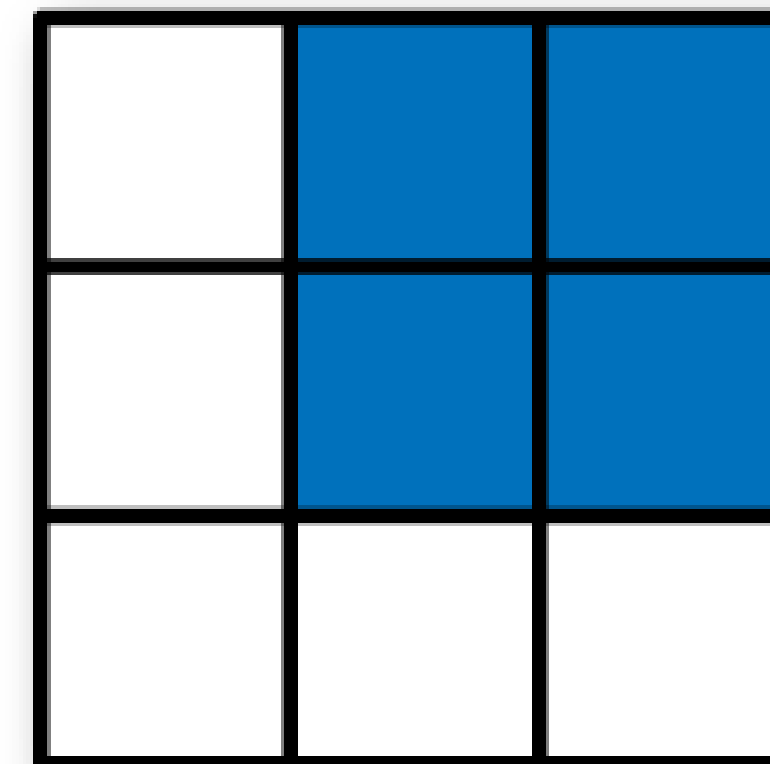


[W. McKinney, Python for Data Analysis]

# 2D Array Slicing

---

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



```
arr[:2, 1:]
```

[W. McKinney, Python for Data Analysis]

# More Reshaping

---

- reshape:
  - `arr2.reshape(4,2)` # returns new view
- resize:
  - `arr2.resize(4,2)` # no return, modifies `arr2` in place
- flatten:
  - `arr2.flatten()` # `array([1.5, 2., 3., 4., 5., 6., 7., 8.])`
- ravel:
  - `arr2.ravel()` # `array([1.5, 2., 3., 4., 5., 6., 7., 8.])`
- flatten and ravel look the same, but ravel is a **view**

# Array Transformations

---

- Transpose
  - `arr2.T` # flip rows and columns
- Stacking: take iterable of arrays and stack them horizontally/vertically
  - `arrh1 = np.arange(3)`
  - `arrh2 = np.arange(3, 6)`
  - `np.vstack([arrh1, arrh2])`
  - `np.hstack([arr1.T, arr2.T])` # ???