CONCEPTS OF PROGRAMMING

MODULE 3
MPCS 50101

THE UNIVERSITY OF
CHICAGO

TRY THIS

MODULE 3
MPCS 50101

THE UNIVERSITY OF
CHICAGO

TRY THIS

TRY THIS

```
# Double your money
number_string = input("> Enter a number: ")
number = int(number_string)

print "The number doubled is ",
print number * 2




# % python workspace.py
# > Enter a number: 3
# The number doubled is  6
```

TRY THIS

```
# Double your money
number_string = input("> Enter a number: ")
number = int(number_string)
print "The number doubled is ",
print number * 2


# > Enter a number: three

Traceback (most recent call last):
  File "workspace.py", line 4, in <module>
    number = int(number_string)
ValueError: invalid literal for int() with base 10: 'three'
```

TRY THIS

TRY THIS

- Exception handling
- An exception is an error that happens during execution of a program
- Python can generate an exception that can be handled, which avoids your program to crash

```
try:
    # Something that might
    # not work
except:
    print "Trouble"
```

TRY THIS

- Surround section of code with `try` and `except` block
- If the code in the try works
  - `except` is skipped
- If the code in the try fails
  - It jumps to the except section

```
try:
    # Something that might
    # not work
except:
    print "Trouble"
```

TRY THIS

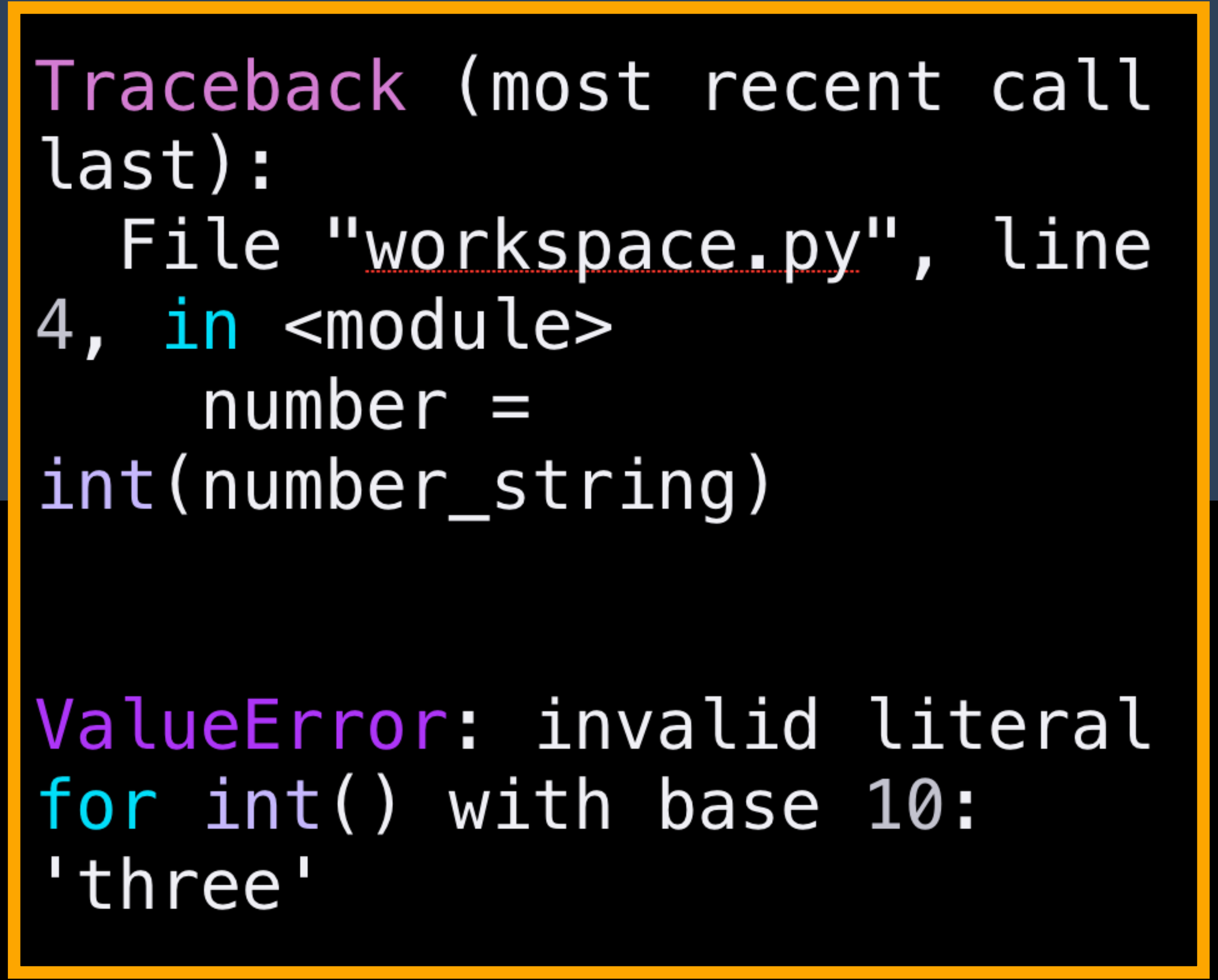
- Exceptions are safer ways for handling errors and special conditions
- Exception built into standard library
- You can write your own

```
Traceback (most recent call last):
  File "workspace.py", line 4, in
<module>
    number = int(number_string)


ValueError: invalid literal for int()
with base 10: 'three'
```

TRY THIS

```
# Double your money
number_string = input("> Enter a number: ")

try:
    number = int(number_string)
    number *= 2
    print(f"The number doubled is {number}")

except:
    print "We couldn't convert the number to an integer."
```

TRY THIS



```
Traceback (most recent call last):
  File "workspace.py", line 4, in <module>
    number = int(number_string)


ValueError: invalid literal for int() with base 10: 'three'
```

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
```

CHECK FOR SPECIFIC ERRORS

TRY THIS

```python
number_string = input("> Enter a number: ")

try:
    number = int(number_string)
except:
    # An exception is raised
    print("We couldn't convert the number to an integer.")
else:
    # No exception was raised
    number *= 2
    print(f"The number doubled is {number}")
finally:
    # This happens no matter what
    print("-- Done --")
```

OPTIONAL;
ALWAYS RUN

TRY THIS

```
number_string = input("> Enter a number: ")

try:
    number = int(number_string)
except:
    # An exception is raised
    print("We couldn't convert the number to an integer.")
else:
    # No exception was raised
    number *= 2
    print(f"The number doubled is {number}")
finally:
    # This happens no matter what
    print("-- Done --")
```

IS THIS THE BEST WAY TO STRUCTURE YOUR CODE?

TRY THIS

```
# Double your money
number_string = input("> Enter a number: ")

try:
    number = int(number_string)
    success = True
except:
    success = False
    print("We couldn't convert the number to an in

if success == True:
    print("The number doubled is ")
    print(number * 2)
```

ISOLATES THE CODE THAT WE ARE "TRYING"

SET A FLAG

STRATEGY FOR CONTINUING EXECUTION EVEN IF FAILS

TRY THIS

```python
# Double your money
number_string = input("> Enter a number: ")

try:
    number = int(number_string)
    success = True
except:
    success = False
    print("We couldn't convert the number to an integer.")

if success == True:
    print("The number doubled is ")
    print(number * 2)
```

TRY THIS

```
success = True

# Double your money
number_string = input("> Enter a number: ")

try:
    number = int(number_string)
except:
    success = False
    print("We couldn't convert the number to an integer.")

if success == True:
    print("The number doubled is ")
    print(number * 2)
```

COLLECTION DATA STRUCTURES

MODULE 3
MPCS 50101

THE UNIVERSITY OF CHICAGO

DATA STRUCTURES

- Data structures allow the storage of data in a consistent manner
- Built-in collection types
    - lists (arrays), dictionaries (hashes), tuples, sets
- Custom types
    - Define your custom data types specific to your programs specification

```
# Built-in collection list
myList = [5, 4, 3, 2, 1]
```

```
# Custom data structure
class Particle:
  def __init__(self velocity,force):
    self.mass = mass
    self.position = position
    self.velocity = velocity
    self.force = force

particle = Particle(2, 3, 3, 8)
```

DATA STRUCTURES

- What is the best type of data structure to use?
- It depends
  - Task
  - Complexity
  - Development time

SOMETIMES SIMPLER IS BETTER

LISTS

LISTS

- Lists are **ordered** collection of objects
- List constants are in square brackets
- Elements separated by commas

```
# List of strings
friends = [ 'Lola', 'Jane', 'Rachel']

# List of integers
favorite_numbers = [ 1, 2, 3, 4, 5]

# Lists can be 'heterogenous'
stuff = [1, "cat", "dog", 34, str()]
```

LISTS

- A list element can be any Python object
  - Even another list

```
# List of strings
cats = [ 'garfield', 'lola', 'scaredy']
dogs = [ 'spot', 'underdog', 'snooopy']

# List of lists
pets = [ cats, dogs ]

>>> pets
[['garfield', 'lola', 'scaredy'],
['spot', 'underdog', 'snooopy']]
```

LIST OF LISTS

LISTS

- Iterating through a list of lists

```
cats = [ 'garfield', 'lola', 'scaredy']
dogs = [ 'spot', 'underdog', 'snooopy']
pets = [ cats, dogs ]


# Loop through the list of lists
for pet in pets:
 # The iteration variable is a list
 for animal in pet:
    print(animal)


>>> [['garfield', 'lola', 'scaredy'],
     ['spot', 'underdog', 'snooopy']]
```

LISTS

```
for pet in pets:
    print(type(pet))

    if isinstance(pet, list):
        for animal in pet:
            print(animal)
```

REMEMBER A LIST CAN CONTAIN DIFFERENT TYPES

```
<type 'list'>
['garfield', 'lola', 'scaredy']
<type 'list'>
['spot', 'underdog', 'snooopy']
```

LISTS

- Access the elements of list by their index

```
cats = [ 'garfield', 'lola', 'scaredy']
dogs = [ 'spot', 'underdog', 'snooopy']
pets = [cats, dogs]

print(pets[0]) )   # cats list
print(pets[0][0])) # garfield
print(pets[0][1])  # lola

print(pets[1])     # dogs list
print(pets[1][1])  # underdog
```

LISTS

- If an index has a negative value, it counts backward from the end

```
cats = [ 'garfield', 'lola', 'scaredy']

print(cats[-1]) # scaredy

print(cats[-2]) # lola

print(cats[-3]) # garfield
```

LISTS

- Any integer expression can be used as an index

```
cats = [ 'garfield', 'lola', 'scaredy']

x = 1
print(cats[x+1] # scaredy)
```

LISTS

- **IndexError** if you try to read or write an element that does not exist

```
cats = [ 'garfield', 'lola', 'scaredy']
index = 4

print(cats[index])
```

```
Traceback (most recent call last):
  File "workspace.py", line 4, in
<module>
    cats[4]
IndexError: list index out of range
```

LISTS

- The `**range()**` function returns a list of numbers that range from zero to one less than the parameter
- Take 1 or 2 parameters

```
for i in range(len(cats)):
    print("#", i,"->",cats[i])
# 0 -> garfield
# 1 -> lola
# 2 -> scaredy

for i in range(0,len(cats)):
    print("#", i,"->",cats[i])
# 0 -> garfield
# 1 -> lola
# 2 -> scaredy

for i in range(1,len(cats)):
    print("#", i,"->",cats[i])
# 1 -> lola
# 2 -> scaredy
```

LISTS

- List are mutable

```
cats = [ 'garfield', 'lola', 'scaredy']
print(cats)
# >>> ['garfield', 'lola', 'scaredy']


# Mutate the value of the value at
# index 1
cats[1] = 'tom'

print(cats)
# >>> ['garfield', 'tom', 'scaredy']
```

LISTS

- Strings are not mutable

```
name = 'Ada'
print(name[0]) # A

name[0] = "B"
```

```
Traceback (most recent call last):
  File "workspace.py", line 11, in
<module>
    name[0] = "B"
TypeError: 'str' object does not
support item assignment
```

LISTS

- Lists can be concatenated using the `+` operator

```
cats = [ 'garfield', 'lola', 'scaredy']

famous_cats = [ 'whiskers', 'grumpy
cat']

# Use the + operator to concatenate
# lists
all_cats = cats + famous_cats

print(all_cats)

# >>> ['garfield', 'lola', 'scaredy',
'whiskers', 'grumpy cat']
```

LISTS

- Combined concatenation and value reassignment

```
cats = [ 'garfield', 'lola', 'scaredy']

famous_cats = [ 'whiskers', 'grumpy
cat']



# Use the += operator to concatenate
# and reassign to original list
# cats = cats + famous_cats
cats += famous_cats


print(cats)
# >>> ['garfield', 'lola', 'scaredy',
'whiskers', 'grumpy cat']
```

LISTS

- Lists can be sliced
    - list[start:stop]

```
cats = [ 'garfield', 'lola', 'scaredy']
famous_cats = [ 'whiskers', 'grumpy
cat']

# Use the + operator to concatenate
# lists
all_cats = cats + famous_cats

print(all_cats)

# >>> ['garfield', 'lola', 'scaredy',
'whiskers', 'grumpy cat']

print(all_cats[2:5])
# >>> ['scaredy', 'whiskers', 'grumpy
cat']
```

LISTS

- A list can be empty

```
drinks = []

print(drinks)  # []
```

LISTS

- List have built-in functions
  - append(item)

```
drinks = []
print(drinks)
# []

drinks.append("Soda")
print(drinks)
# ['Soda']

drinks.append("Wine")
print(drinks)
# ['Soda', 'Wine']

drinks.append("Beer")
print(drinks)
# ['Soda', 'Wine', 'Beer']
```

LISTS

- List have built-in functions
  - extend(list)

```
drinks = []
print(drinks)  # []


drinks.append("Soda")
print(drinks)
# ['Soda']


more_drinks = ["Wine", "Beer"]
drinks.extend(more_drinks)
print(drinks)
# ['Soda', 'Wine', 'Beer']
```

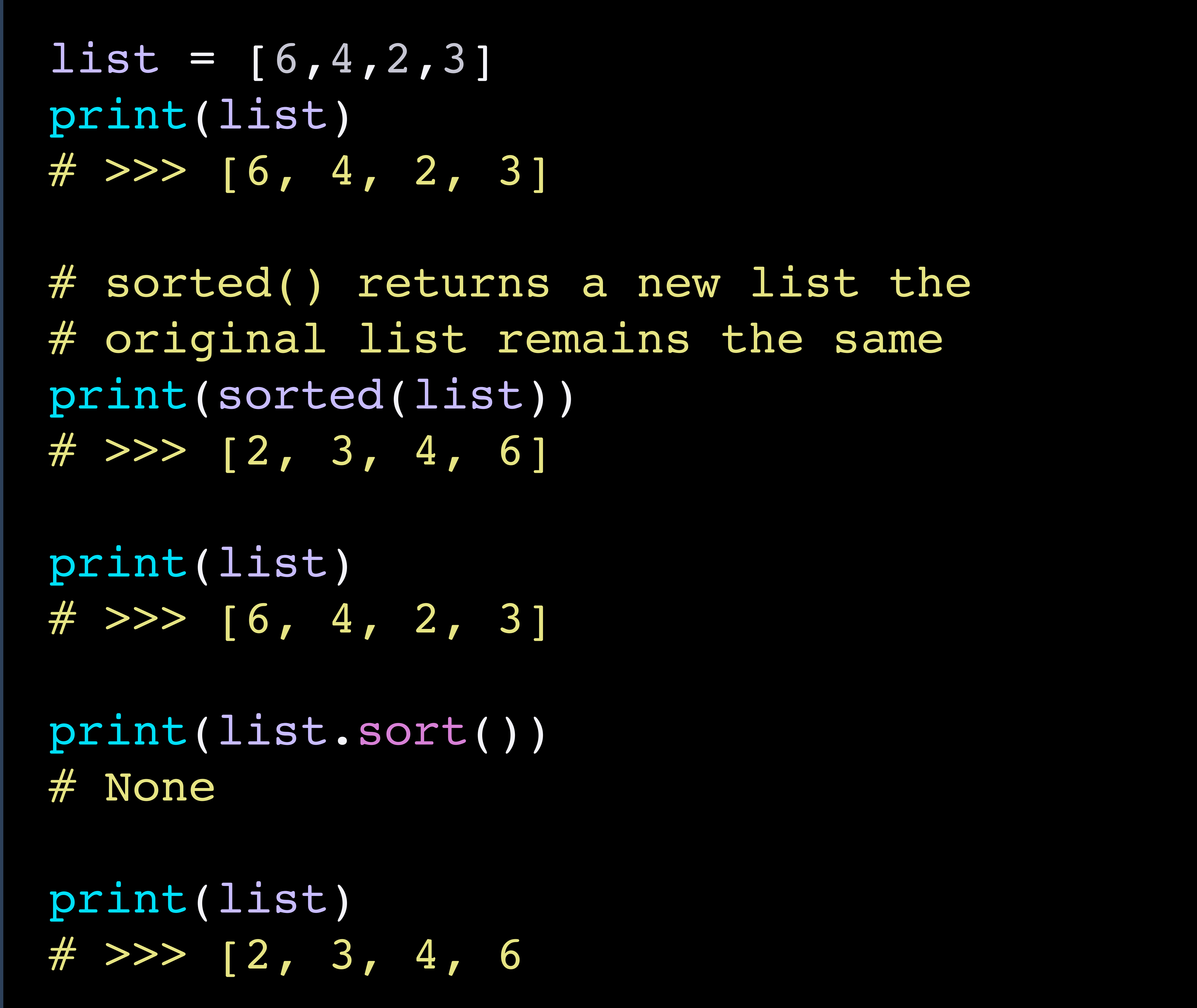
LISTS

- List have built-in functions
    - sort()

```
print(drinks)
# ['Soda', 'Wine', 'Beer']

drinks.sort()

print(drinks)
# ['Beer', 'Soda', 'Wine']
```

LISTS

- sorted(list) vs list.sort()
- Pay attention to returned values of functions



```
list = [6,4,2,3]
print(list)
# >>> [6, 4, 2, 3]

# sorted() returns a new list the
# original list remains the same
print(sorted(list))
# >>> [2, 3, 4, 6]

print(list)
# >>> [6, 4, 2, 3]

print(list.sort())
# None

print(list)
# >>> [2, 3, 4, 6
```

LISTS

```
drinks = []
print(drinks)  # []

drinks.append("Soda")
print(drinks) # ['Soda']

more_drinks = ["Wine", "Beer"]
drinks += more_drinks
print(drinks)
# ['Soda', 'Wine', 'Beer']

drinks.insert(0,'Lemonade')
print(drinks)
# ['Lemonade', 'Soda', 'Wine', 'Beer']
```

INSERT(INDEX,VALUE)

LISTS

```
drinks = ["Soda", "Wine", "Beer", "Lemonade"]
print(drinks)
# ['Soda', 'Wine', 'Beer', 'Lemonade']

del drinks[0]
print(drinks)
# ['Wine', 'Beer', 'Lemonade']

del drinks [0:2]
print(drinks)
# ['Lemonade']
```

DEL LIST[INDEX] REMOVES THE ITEM AND DOES NOT RETURN IT

LISTS

```
drinks = ["Soda", "Wine", "Beer", "Lemonade"]
del drinks[0]

print drinks
# ['Wine', 'Beer', 'Lemonade']

removed_item = drinks.pop()
print removed_item
# Lemonade

print drinks
# ['Wine', 'Beer']
```

POP RETURNS THE ELEMENT REMOVED

LISTS

```
drinks = ["Soda", "Wine", "Beer", "Lemonade"]
print(drinks)
# ['Lemonade', 'Soda', 'Wine', 'Beer']

drinks.remove('Wine')
drinks.remove('Beer')
print(drinks)
# ['Lemonade', 'Soda']
```

REMOVE(ELEMENT) REMOVES THE ELEMENT

LISTS

- dir()
    - attempt to return a list of valid attributes (properties and methods) for that object

```
>>> x = list()
>>> type(x)

<type 'list'>
>>> y = []

>>> type(y)
<type 'list'>

>>> dir(y)
['append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse',
'sort']
```

LISTS

__X__ ARE SPECIAL FUNCTIONS USED INTERNALLY BY PYTHON

```
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

LISTS

```
>>> s = 'hi'
>>> type(s)
<type 'str'>
>>> dir(s)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

LISTS

- Lists can be function arguments

```
def add_them_up(numbers):
    # Take a list of numbers, sum
    # them and return the total
    total = 0
    for number in numbers:
        total = total + number
    return total


scores = [3, 41, 12, 9, 74, 15]
print(add_them_up(scores))
```

LISTS

- Functions can return a list

```
def first_and_last(numbers):
    # Return the first and last
    # item of a list
    first = numbers[0]
    last = numbers[-1]
    first_last = [first, last]
    return first_last


scores = [3, 41, 12, 9, 74, 15]
print(first_and_last(scores))
# >>> [3, 15]
```

LISTS

- There are a number of functions built into Python that take lists as parameters

```
nums = [3, 41, 12, 9, 74, 15]

print(len(nums))             # 6
print(max(nums))             # 74
print(min(nums))             # 3
print(sum(nums))             # 154
print(sum(nums)/len(nums))   # 25
```

TUPLES

TUPLES

- Tuples are another kind of sequence that functions like a list

- A tuple is a fixed size grouping of elements

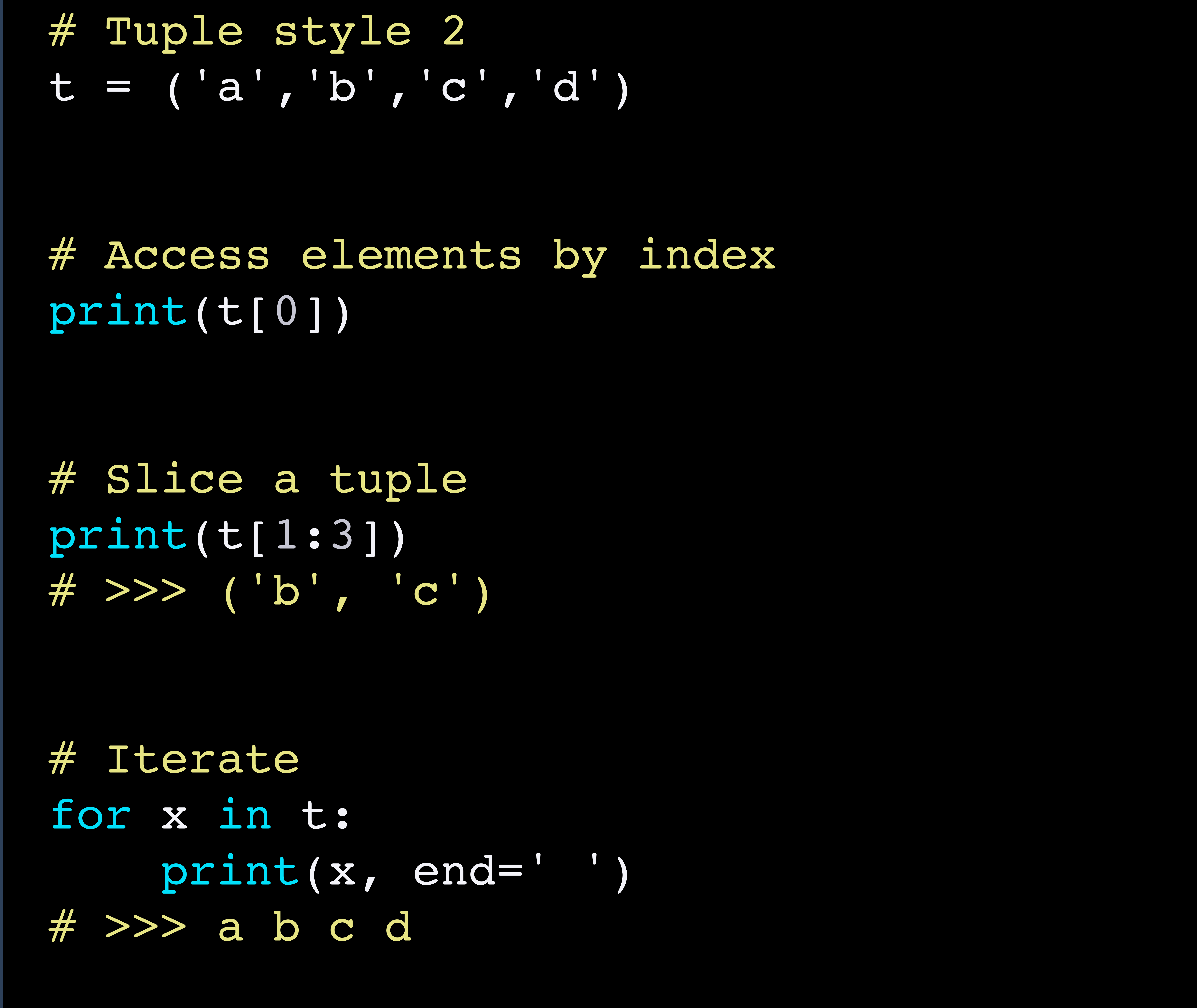
  - Elements which are indexed starting at 0

```
# Tuple style 1
t = 'a','b','c','d'

print(type(t))
# >>> <type 'tuple'>
```

```
# Tuple style 2
t = ('a','b
','c','d')

print(type(t))
# >>> <type 'tuple'>
```

TUPLES

- Tuples behave likes lists



```
# Tuple style 2
t = ('a','b','c','d')


# Access elements by index
print(t[0])


# Slice a tuple
print(t[1:3])
# >>> ('b', 'c')


# Iterate
for x in t:
    print(x, end=' ')
# >>> a b c d
```

TUPLES

- Tuples are comparable
- If the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ

```
>>> (0, 1, 2) < (5, 1, 2)
True

>>> (0, 1, 2000000) < (0, 3, 4)
True

>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True

>>> ( 'Jones', 'Sally') == ('Jones', 'Sam')
False
```

TUPLES

- Tuples are not mutable

```
t = ('a','b','c','d')
t[2] = 'Z'


Traceback
  File "workspace.py", line 9, in
<module>
    t[2] = 'Z'
TypeError: 'tuple' object does not
support item assignment
```

TUPLES

- Tuples do not share all the functions as list

```
>>> x = (3, 2, 1)

>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no
attribute 'sort'

>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no
attribute 'append'

>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no
attribute 'reverse'
```

TUPLES

- Tuples do not share all the functions as list

```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse',
'sort']
```

```
>>> t = tuple()
>>> dir(t)
['count', 'index']
```

TUPLES

- Why then?
  - Simpler and more efficient in terms of memory use and performance than lists
  - Useful for “temporary variables”

```
def biggest_and_smallest():
    """
    """
    return (big,small)
```

TUPLES

- Why then?
  - Data definition
    - (x, y, z) for a coordinate
    - (long, lat) for GPS position

```
xyz = (1,2,3)

coordinates = (long, lat)
```

TUPLES

- Tuples are convenient

```python
def first_and_last(numbers):
    # Return the first and last
    # item of a list
    first = numbers[0]
    last = numbers[-1]
    first_last = [first, last]
    return first_last



scores = [3, 41, 12, 9, 74, 15]
print(first_and_last(scores))
# >>> [3, 15]
```

TUPLES

- Tuples have a couple of neat tricks
- We can also put a tuple on the left-hand side of an assignment statement
- We can even omit the parentheses

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred

>>> (a, b) = (99, 98)
>>> print(a)
99

>>> a, b = (99, 98)
>>> print(a)
99
```

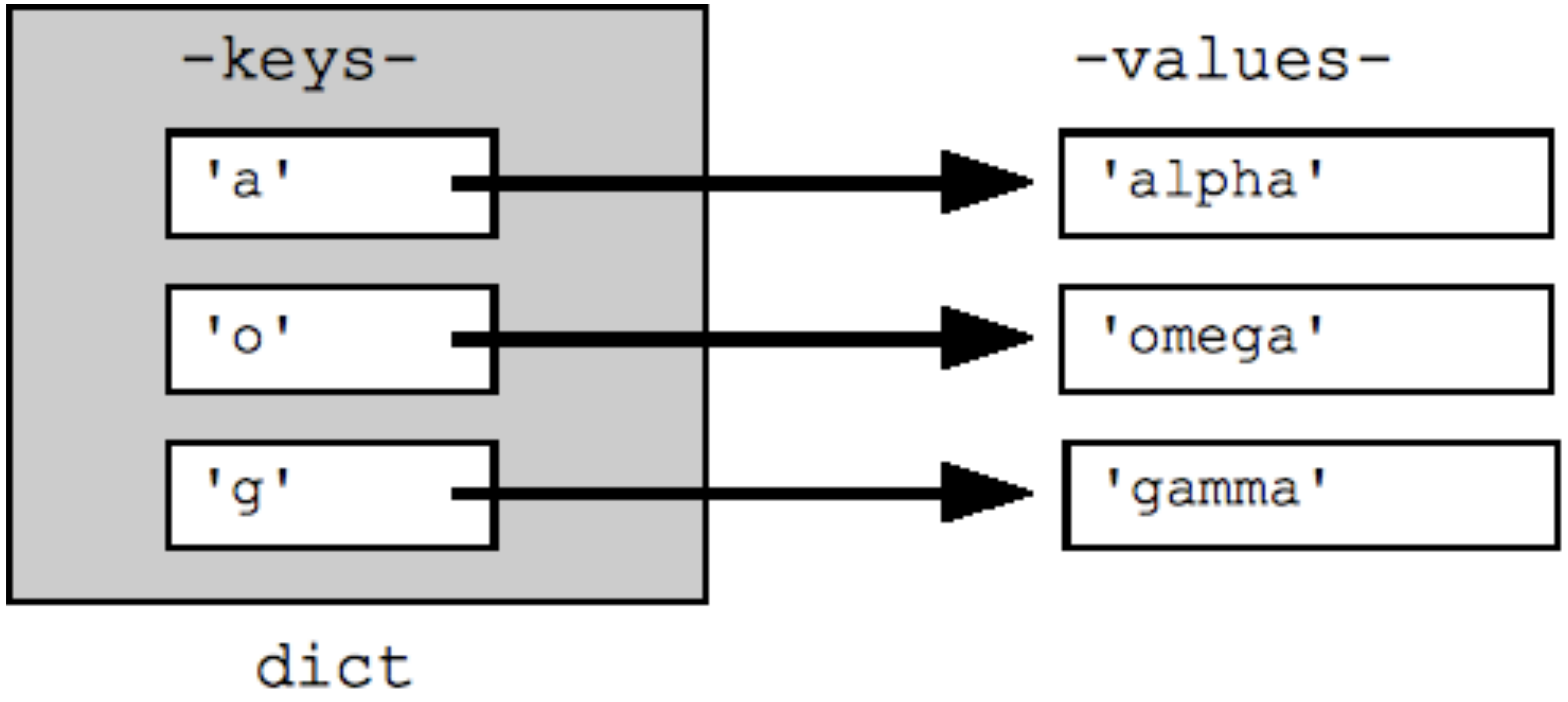
TUPLES

```
# Swap the variables a and b
>>> temp = a
>>> a = b
>>> b = temp


# Swap the variables a and b
>>> a,b = b,a
```

DICTIONARIES

DICTIONARIES

- Dictionaries are Python's most powerful data collection
  - Store values associated with a key
  - Perform "lookup" operations
- Dictionaries have different names in different languages
  - Associative Arrays - Perl / PHP
  - Properties or Map or HashMap - Java
  - Property Bag - C# / .Net



KEY VALUE PAIRS

DICTIONARIES

- Dictionaries are like lists except that they use `keys` instead of numbers to look up values

```
person = dict()
person['firstname'] = 'Bruce'
person['lastname'] = 'Wayne'
person['nickname'] = 'Batman'
person['enemies'] =['Joker','Catwoman']
person['age'] = 40
```

```
# >> {'lastname': 'Wayne',
#      'age': 40,
#      'nickname': 'Batman',
#      'firstname': 'Bruce',
#      'enemies': ['Joker', 'Catwoman']
#.     }
```

DICTIONARIES

```python
person['firstname'] = 'Bruce'
person['lastname'] = 'Wayne'



# Lookup the value using the string key
print(person['firstname']) # Bruce
print(person['lastname'])  # Wayne
```

DICTIONARIES

- Dictionary literals use curly braces and have a list of 'key : value' pairs
- You can make an empty dictionary using empty curly braces

```
hero = { 'firstname': 'Clark',
         'lastname': 'Kent',
         'age': 30 }

print(hero)
# >>> {'lastname': 'Kent', 'age': 30,
'firstname': 'Clark'}


# Create an empty dictionary
empty_hero = {}
# >>> {}
```

DICTIONARIES

```
vehicles = { 'bus' : 1 , 'car' : 42, 'van': 10, 'rv': 2}
print vehicles['motorcycle']
```

```
Traceback (most recent call last):
  File "workspace.py", line 20, in <module>
    print vehicles['motorcycle']
KeyError: 'motorcycle'
cles['motorcycle']
KeyError: 'motorcycle'
```

KEYERROR

DICTIONARIES

```
vehicles = { 'bus' : 1 , 'car' : 42, 'van': 10, 'rv': 2}



# Test if a key is in a dictionary with `in`
if 'motorcycle' in vehicles:
    print "Motorcycle"
else:
   print "No motorcycle"
```

DICTIONARIES

- Pattern of checking if key exists and then looking it up

```
# Retrieve the count of motorcycles
# from the dictionary
count = 0
if 'motorcyle' in vehicles:
    count = vehicles['motorcyle']
else:
    count = 0
```

- Built-in function `get()` provides this

```
# Use `get` to test and return default
# value
count = vehicles.get('motorcycle',0)
```

DICTIONARIES

- Delete key-value pairs using `del`

```
# del with variables
var = 6
del var # var no more!


# del in lists
list = ['a', 'b', 'c', 'd']
del list[0]    ## Delete first element
del list[-2:] ## Delete last two
print(list) ## ['b']


# del in dictionaries
dict = {'a':1, 'b':2, 'c':3}
del dict['b']
print(dict)
## >>> {'a':1, 'c':3}
```

DICTIONARIES

- Dictionaries are unordered

```
hero = { 'firstname': 'Clark',
         'lastname': 'Kent',
         'age': 30 }


print(hero)

# >>> {'lastname': 'Kent',
#        'age': 30,
#        'firstname': 'Clark'}
```

DICTIONARIES

- Loop through all the entries in a dictionary
- Goes through all of the keys in the dictionary and looks up the values

```python
vehicles = { 'bus' : 1 , 'car' : 42,
             'van': 10, 'rv': 2}


# The iteration variable is lookup key
for vehicle in vehicles:
    print(vehicle)
# >>> bus, rv, van, car


for vehicle in vehicles:
    print(vehicle, vehicles[vehicle])

# >>> bus 1
# >>> rv 2
# >>> van 10
# >>> car 42
```

DICTIONARIES

- Different ways to access keys/ values in a dictionary

```
print list(vehicles)
# ['bus', 'rv', 'van', 'car']

print vehicles.keys()
# ['bus', 'rv', 'van', 'car']

print vehicles.values()
#[1, 2, 10, 42]

print vehicles.items()
# [('bus', 1), ('rv', 2), ('van', 10),
('car', 42)]
```

DICTIONARIES

- .items() returns a tuple as the iteration variable

- Shortcut to assign them directly to a pair of iteration variables

```
# Using .items() returns a tuple
print vehicles.items()

# >>> [('bus', 1), ('rv', 2), ('van',
10), ('car', 42)]


for key,value in vehicles.items():
    print key,"->",value
# bus -> 1
# rv -> 2
# van -> 10
# car -> 42
```

MOST USEFUL

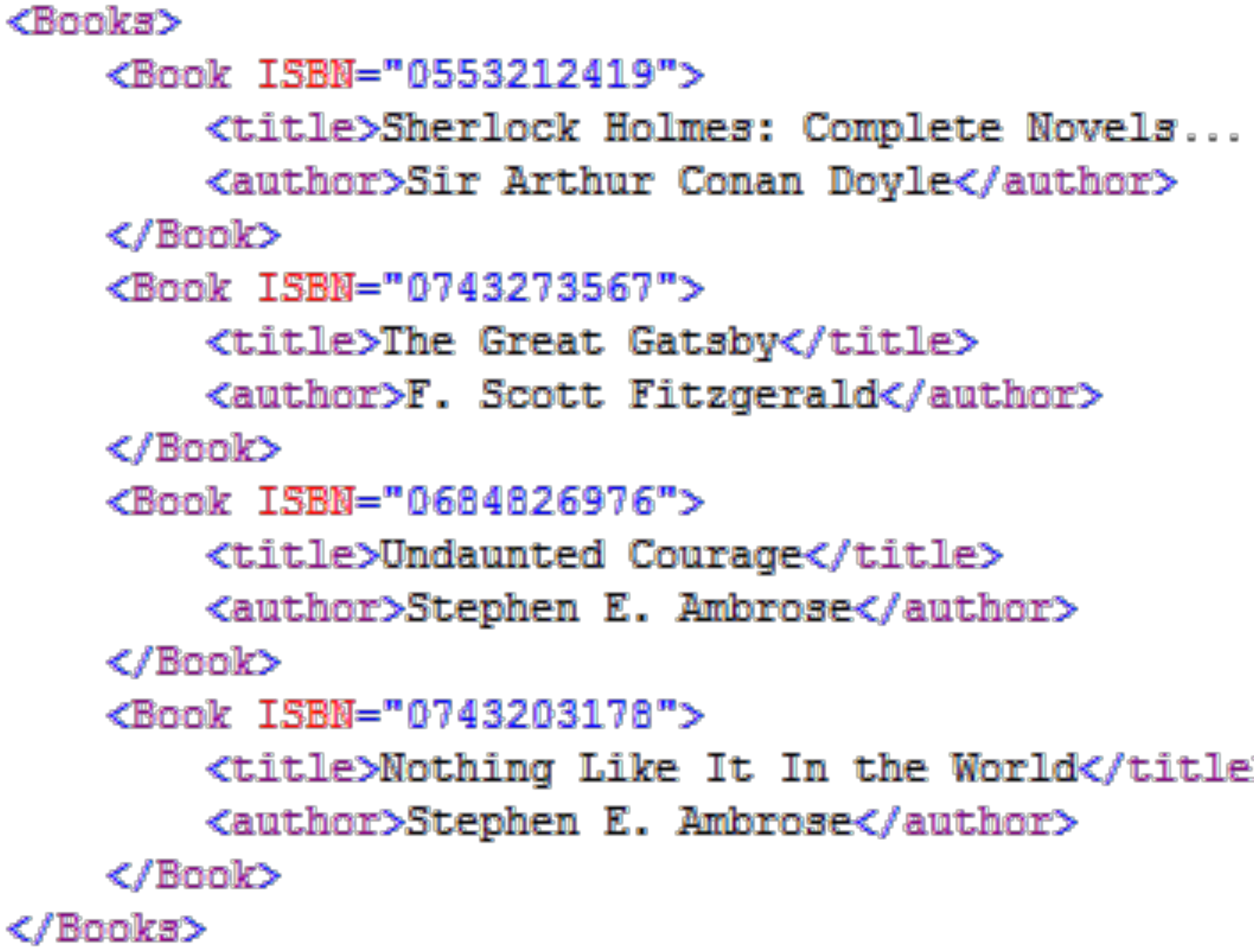
FILES AND PARSING

MODULE 3
MPCS 50101

THE UNIVERSITY OF
CHICAGO

FILES AND PARSING

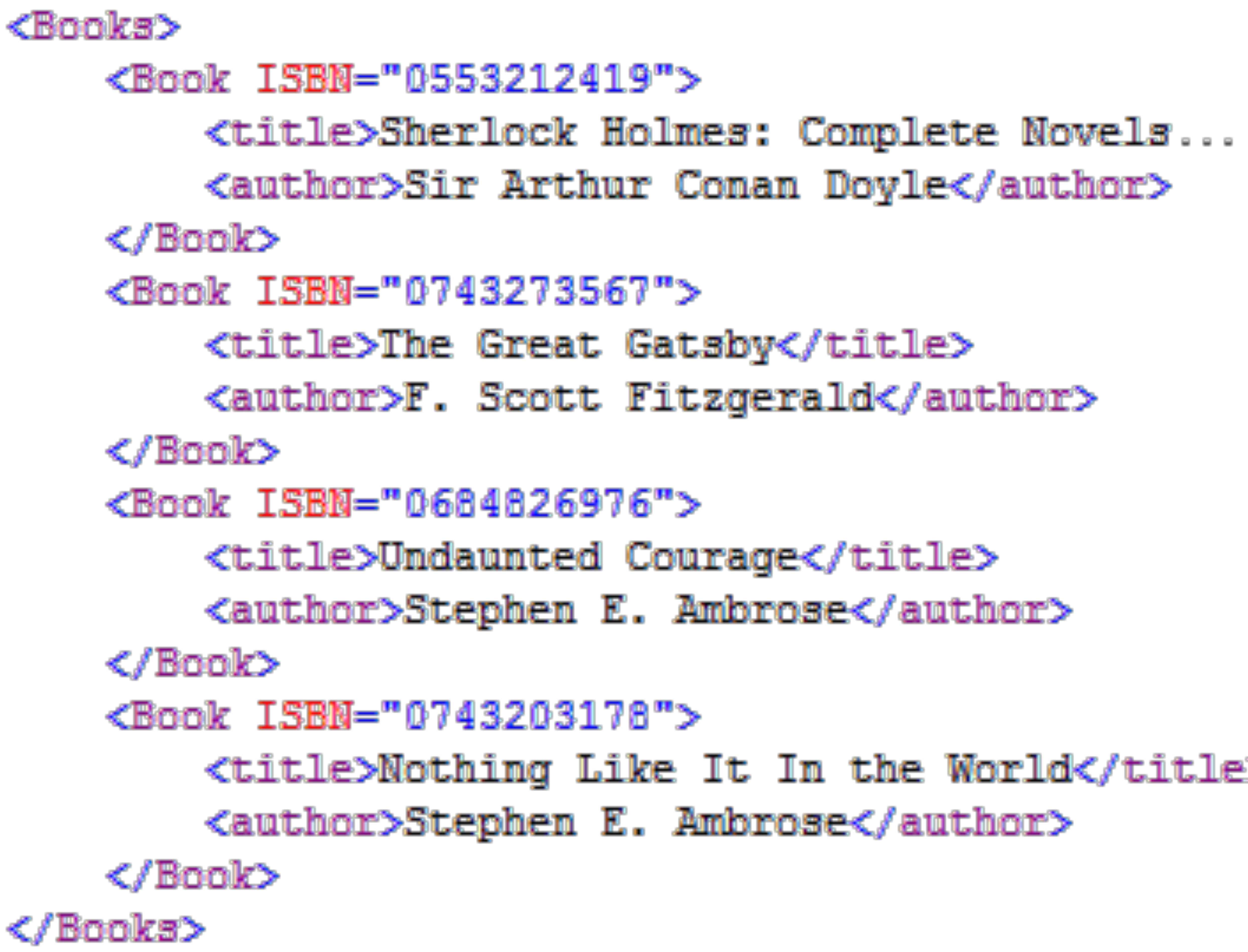
- Applications need to persist data between sessions
- Reading and writing data to disk



```
<Books>
    <Book ISBN="0553212419">
        <title>Sherlock Holmes: Complete Novels...
        <author>Sir Arthur Conan Doyle</author>
    </Book>
    <Book ISBN="0743273567">
        <title>The Great Gatsby</title>
        <author>F. Scott Fitzgerald</author>
    </Book>
    <Book ISBN="0684826976">
        <title>Undaunted Courage</title>
        <author>Stephen E. Ambrose</author>
    </Book>
    <Book ISBN="0743203178">
        <title>Nothing Like It In the World</title
        <author>Stephen E. Ambrose</author>
    </Book>
</Books>
```

FILES AND PARSING

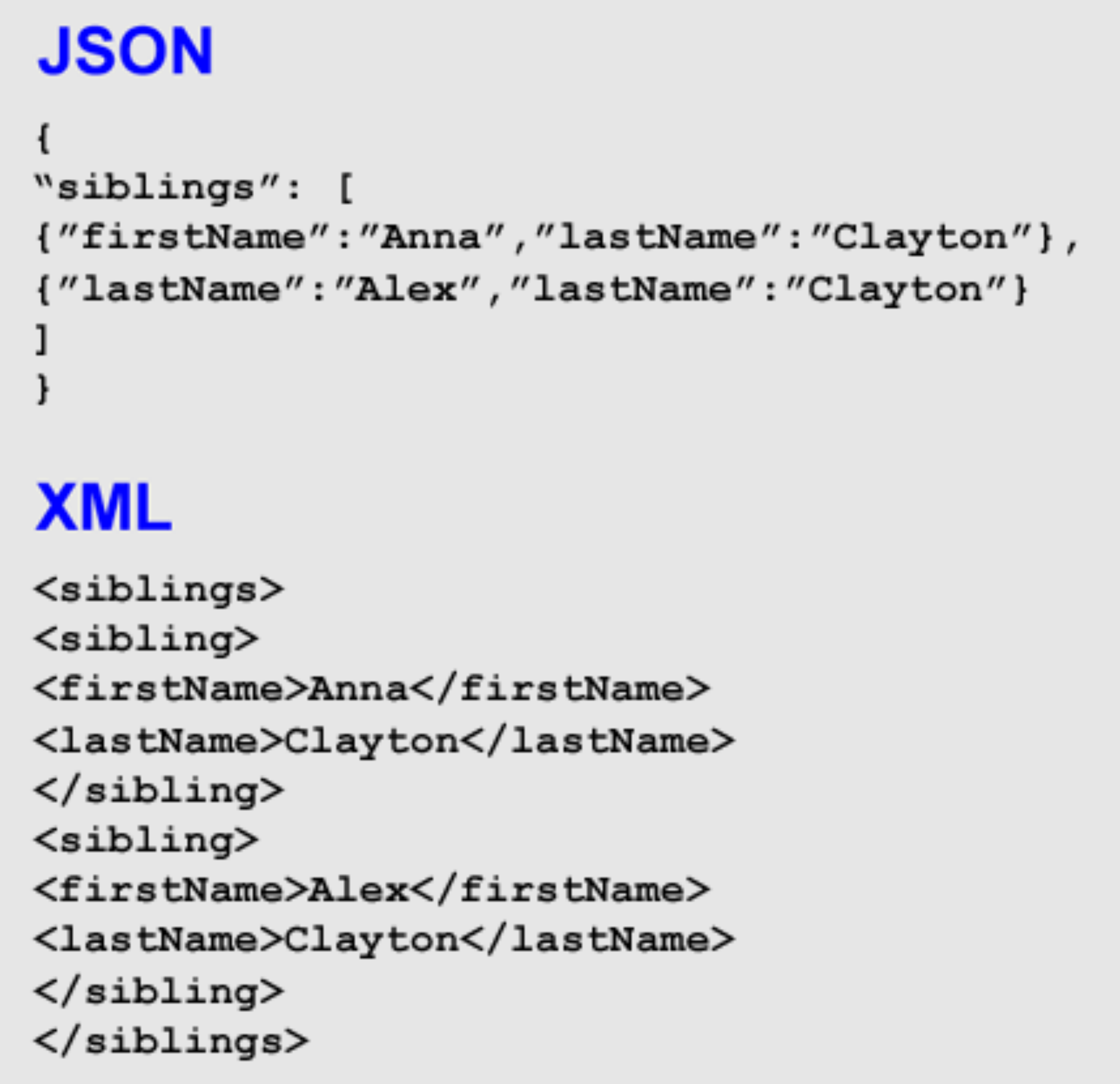
- Files contain different types of data
  - Structured format (csv, XML, JSON, HTML)
  - Text (.txt)
  - Binary (.docx, .sql)
- An application can use any of these file types to save data



```
<Books>
    <Book ISBN="0553212419">
        <title>Sherlock Holmes: Complete Novels...
        <author>Sir Arthur Conan Doyle</author>
    </Book>
    <Book ISBN="0743273567">
        <title>The Great Gatsby</title>
        <author>F. Scott Fitzgerald</author>
    </Book>
    <Book ISBN="0684826976">
        <title>Undaunted Courage</title>
        <author>Stephen E. Ambrose</author>
    </Book>
    <Book ISBN="0743203178">
        <title>Nothing Like It In the World</title
        <author>Stephen E. Ambrose</author>
    </Book>
</Books>
```

FILES AND PARSING

- Different formats have different properties
  - Size
  - Privacy
  - Human readable



JSON

```
{
"siblings": [
{"firstName":"Anna","lastName":"Clayton"},
{"lastName":"Alex","lastName":"Clayton"}
]
}
```

XML

```
<siblings>
<sibling>
<firstName>Anna</firstName>
<lastName>Clayton</lastName>
</sibling>
<sibling>
<firstName>Alex</firstName>
<lastName>Clayton</lastName>
</sibling>
</siblings>
```

READING FILES

READING FILES

- Python has built-in functions to help read text-based files
- Reading some file types may require special libraries
    - Write your own

```python
# Open the file
f = open('infile.txt', 'r')

## Iterate over lines of
#  the file
for line in f:
  print(line)
f.close()
```

READING FILES

- Tell Python which file we are going to work with and what we will be doing with the file
- This is done with the **open()** function
- Returns a “file handle”
  - A variable used to perform operations on the file

```
# Open the file
f = open('infile.txt', 'r')

## Iterate over lines of the file
for line in f:
  print(line)
f.close()
```

READING FILES

- handle = open(filename, mode)
  - Returns a file handle
  - Filename is a string
  - Mode is optional
    - 'r' if we are planning to read the file
    - 'w' if we are going to write to the file

```
# Open the file for reading
f = open('./speech.txt', 'r')
```

CURRENT DIRECTORY

```
# Write to a file
f = open('./speech.txt', 'w')
```

READING FILES

- Good place to use try/except

```
# You need to check if a files is
# present before opening it
f = open('./missing.txt', 'r')
```

```
Traceback (most recent call last):
  File "workspace.py", line 2, in
<module>
    f = open('./speedch.txt', 'r')
IOError: [Errno 2] No such file or
directory: './missing.txt'
```

READING FILES

- A file handle open for reading can be treated as a sequence of strings
- Each line in the file is a string in the sequence

```
# Open the file
f = open('infile.txt', 'r')

## Iterate over lines of the file
for line in f:
  print(line, end='')
f.close()
```

CLOSE FILE HANDLES WHEN DONE

READING FILES

- Files have special (invisible) characters used in formatting
  - `\n` - "newline” to indicate when a line ends
  - `\t` - tab

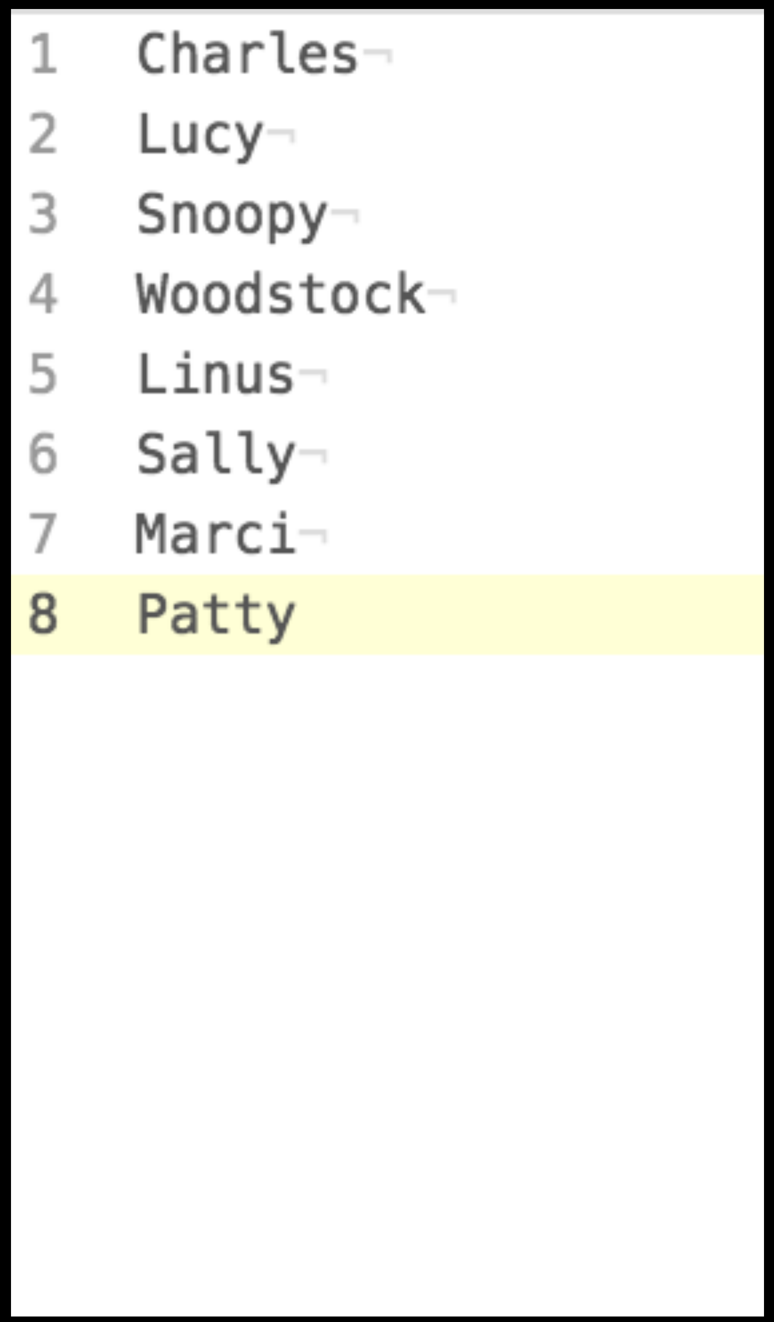
```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'

>>> print stuff
Hello
World!
```

```
>>> stuff = 'X\nY'
>>> print stuff
X
Y
>>> len(stuff)
3
```

READING FILES

- Pay attention to newline characters when printing
- You are reading them in from the file

```
f = open('./names.txt', 'r')
for line in f:
    print(line)
f.close()



# Charles
#
# Lucy
#
# Snoopy
```



```
Charles¬
Lucy¬
Snoopy¬
Woodstock¬
Linus¬
Sally¬
Marci¬
Patty
```

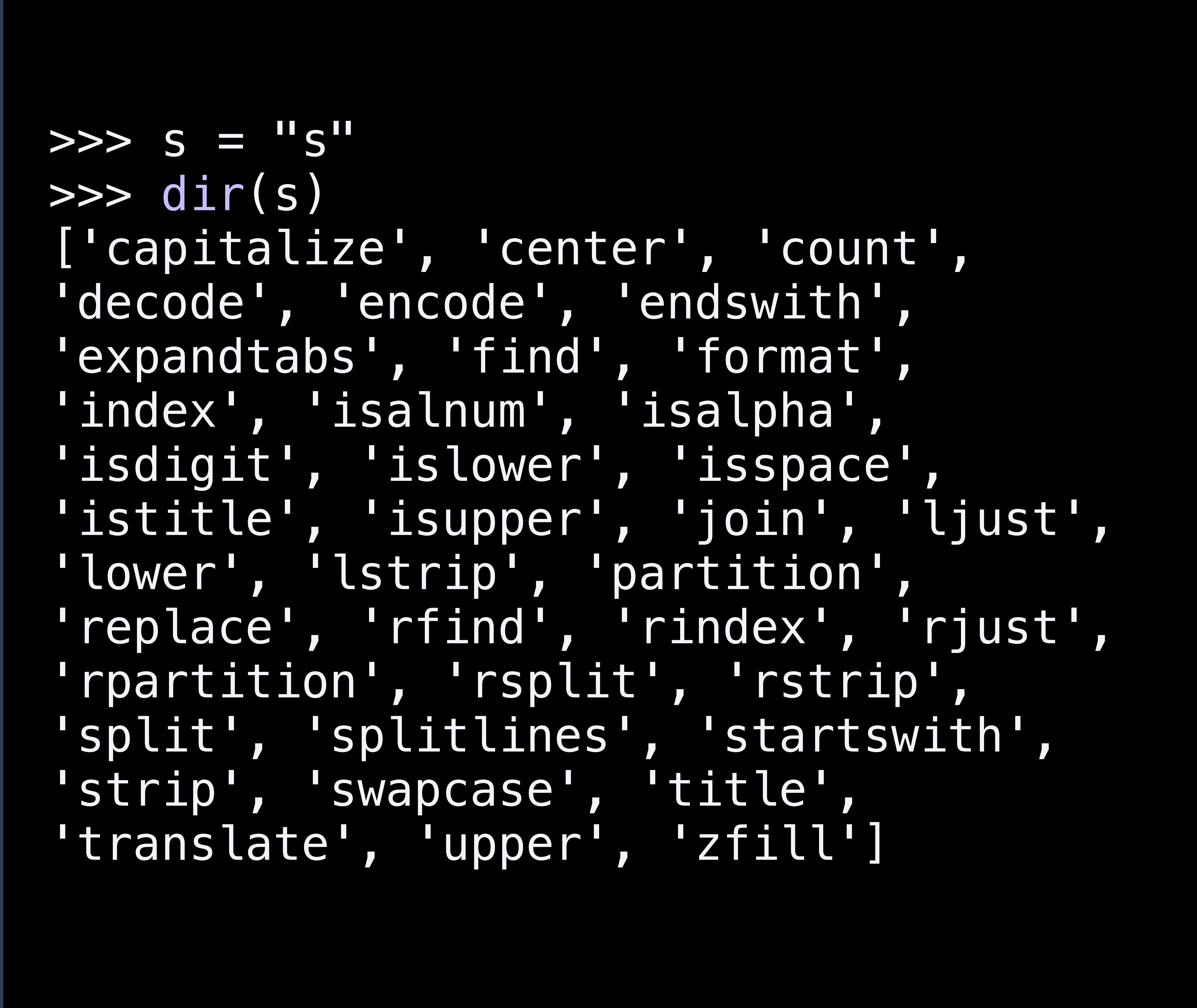
READING FILES

- We can read the whole file into a single string
    - \n are read in as well

```python
# Read the entire file into
# a variable
f = open('./names.txt', 'r')

entire_file = f.read()


print(len(entire_file))
# >> 54
```

READING FILES

- Clean up your data using string functions
- 'Whitespace' means characters you can not see
    - \t,\n,\ ,etc.



```
>>> s = "s"
>>> dir(s)
['capitalize', 'center', 'count',
'decode', 'encode', 'endswith',
'expandtabs', 'find', 'format',
'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition',
'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

READING FILES

```
# Echo the contents of a file
f = open('./names.txt', 'r')

for line in f:

    # Remove whitespace
    clean_line = line.strip()

    # Make case uniform
    clean_line = clean_line.lower()
    print(clean_line)

f.close()
```

READING FILES

```
# Split() splits a string into a list at a " "
>>> y = 'the quick brown fox jumped over the yellow dog'
>>> z = y.split()
>>> z
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the',
'yellow', 'dog']


# Split(',') splits a string on a comma
>>> y = "1,2,3,4,5,6,7,8,9"
>>> z = y.split(',')
>>> z
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

READING FILES

- Cleaning up data is a fundamental problem in computer science
- There are many tools and workflows to accomplish a task
    - Sometimes it is easier to clean up your data before inputing it into a program
    - Sometimes you will write two programs (one to clean, one to process)

WRITING FILES

WRITING FILES

- Python has built-in functions to `write()` files

```
# Open the file
fhandle = open('out.txt', 'w')

# Write a string
fhandle.write("Hello File")

# Close the file
fhandle.fclose()
```

WRITING FILES

- `write()` stops in place
- No newline `\n` by default

```
# Open the file
fhandle =  open('out.txt', 'w')

# Write a string
fhandle.write("Hello File\n")

# Close the file
fhandle.fclose()
```

EXPLICIT

WRITING FILES

- If the file already exists, it will overwrite it
- Use the `a` to append to an existing file

```
# Open a file to append text
fhandle =  open('out.txt', 'a')

# Write a string immediately
# after the last entry
fhandle.write("Hello File")

# Close the file
fclose()
```

WRITING FILES

```
fhandle = open('out.txt', 'w')
fhandle.write(str(1)) # Works


fhandle.write(1)

Traceback (most recent call last):
  File "writing_files.py", line 2, in
<module>
    fhandle.write(1)
TypeError: expected a string or other
character buffer object
```

- Argument of `write()` has to be a string
- Need to convert all other values

WRITING FILES

- Use string formatting to save yourself from having to convert everything to strings

```
fhandle =  open('out.txt', 'a')

# String formatting
line = "%s %d %d %d" % ("String",1,2,3)

# Write string to file
fhandle.write(line)

fhandle.close()
```

WRITING FILES

```
'%s %s' % ('one', 'two')    # one two
'%10s' % ('test',)          # "      test"
'%-10s' % ('test',)         # "test      "
'%.3f' % (3.1415927,)       # 3.142
'%06.2f' % (3.1415927,)     # 003.14
```

WRITING FILES

- Important modules for filenames and paths
- `os` exposes operating system functionality

```
import os
dir(os)     # Operating system


os.getcwd() # Current working dir
```

WRITING FILES

```
>>> os.cwd()
# /Users/tbinkowski/Google Drive/g-Teaching/uchicago.codes/
lectures/session4

>>> os.path.abspath('c.py')
# /Users/tbinkowski/Google Drive/g-Teaching/uchicago.codes/
lectures/session4/c.py

>>> os.path.isdir('../')
# True

>>> os.listdir('./')
# ['workspace.py', 'writing_files.py']
```

DOCSTRINGS

MODULE 3
MPCS 50101

THE UNIVERSITY OF
CHICAGO

DOCSTRINGS

- A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition

```
def say_hello():
    """Prints a greeting to console"""
    print "Hello class"
```

DOCSTRINGS

- docstrings become the __doc__ special attribute of that object

```
def say_hello():
    """Prints a greeting to console"""
    print "Hello class"



# >>> import greeting
# >>> greeting.say_hello.__doc__
#    'Prints a greeting to console'

# >>> help(greeting.say_hello)
```

DOCSTRINGS

- docstrings follow specific format
- Written as command

```
# Good
def func1():
 """Do this and return value"""

# Good
def func1():
 """Return the value"""

# Bad
def function3():
  """ function3() """
```

DOCSTRINGS

- Multi-line docstrings

```python
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part
    """

    if imag == 0.0 and real == 0.0:
        return complex_zero
```

DOCSTRINGS

greeting.py
greeting.pyc
links.md
simple_test.py
simple.py
simple.pyc
workspace.py
writing_files_out.txt
writing_files.py

```
    they should be listed as ``*args`` and ``**kwargs``.

    The format for a parameter is::

        name (type): description
            The description may span multiple lines. Following
            lines should be indented. The "(type)" is optional.

            Multiple paragraphs are supported in parameter
            descriptions.

    Args:
        param1 (int): The first parameter.
        param2 (:obj:`str`, optional): The second parameter. Defau
            Second line of description should be indented.
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.

    Returns:
        bool: True if successful, False otherwise.

        The return type is optional and may be specified at the begi
        the ``Returns`` section followed by a colon.
```

HTTP://SPHINXCONTRIB-NAPOLEON.READTHEDOCS.IO/EN/LATEST/EXAMPLE_GOOGLE.HTML

THE END

MODULE 3
MPCS 50101

THE UNIVERSITY OF
CHICAGO