LECTURES WEEK 3

AGENDA

week	subject	book	week	subject	book
1	Python features	1	3	any & all	19
	running Python	1		range, zip & enumerate	12
	dynamic binding	2		higher-order functions	16
	Python statements	1		classes and OOP	15..18
	printing stuff	2,3		exceptions	14
	Python types	1		assert	16
	numbers	1		file access	14
	strings	8		working with CSV and JSON	-
	control statements	7		coding style	-
	lists	10			
2	tuples	12	4	case: word histogram	13
	dictionaries	11		recursion	5
	sets	19		case: solving Numbrix	-
	functions	6		PySerial	-
	scope/visibility	11		tkinter GUI-toolkit	-
	comprehension	19		web-programming	
	modules and packages	14			

AGENDA

- **any & all**
- range, zip & enumerate
- higher-order functions
- classes and OOP
- exceptions
- assert
- file access
- working with CSV and JSON
- coding style

ANY AND ALL

- to evaluate a sequence of Boolean values
  - any returns True if any of the values are True
  - all returns True if all of the values are True
- often used with generator expressions ( )
  - a generator is a (convenient) type of iterator

```
>>> any([False, False, True])
True
>>> all([False, False, True])
False
>>> any(letter == 't' for letter in 'monty')
True
>>> all(x < 5 for x in [1,2,3,4])
True
```

WHAT WILL BE PRINTED?

```
 1  cnt = 0
 2
 3  def f():
 4      global cnt
 5      cnt = cnt + 1
 6      print(cnt)
 7      if cnt == 11:
 8          return True
 9      else:
10          return False
11
12  if any((f() for i in range(100))):
13      print ("Done")
```

```
1
2
3
4
5
6
7
8
9
10
11
Done
```

AGENDA

- any & all
- **zip & enumerate**
- higher-order functions
- classes and OOP
- exceptions
- file access
- working with CSV and JSON
- coding style

AVOID COUNTING THINGS

- use range, zip and enumerate
- **`range([start], stop[, step])`**
- returns a list of numbers which is generally used to iterate over in for loops
- result is not a list, but a generator, you have to do list(range(5))

WHAT IS A GENERATOR?

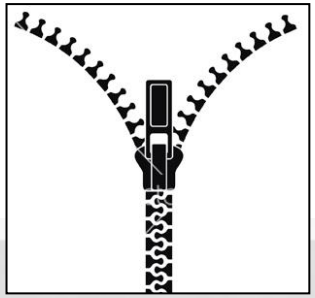
- normal functions return a single value using return
- a generator returns an object (called iterator) on which you can call next
- gives better performance: lazy (on demand) generation of values
  - `sum_of_first_n = sum(list(range(10000)))` : will first build a complete list before passing it to sum
  - `sum_of_first_n = sum(range(10000))` : will pass numbers one-by-one, avoids building a huge lists in memory

WHAT IS A GENERATOR?

```
>>> def simple_generator():
        n = 0
        while True:
            yield n
            n = n + 1


>>> g = simple_generator()
>>> next(g)
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
```

ZIP



- zip : visit multiple sequences in parallel
  - using multiple indexes
  - result is not a list, but a generator, you have to do list(zip(L1, L2)
- if the lists are different lengths, zip stops as soon as one list ends

QUESTIONS

a) L1 = [1,2,3,4], L2 = [5,6,7,8,9,10]
   result: [(1, 5), (2, 6), (3, 7), (4, 8)]

b) L1 = [1,2,3,4], L2 = [5,6,7,8,9,10]
   result: [(1, 5, 6), (2, 6, 8), (3, 7, 10), (4, 8, 12)]

c) k = ['bier', 'ei', 'toast', 'spek']
   v = [1, 3, 5, 7]
   result: {'bier': 1, 'ei': 3, 'toast': 5, 'spek': 7}

ANSWERS

a) `list(zip(L1, L2))`

b) `[(x, y, x+y) for x,y in zip(L1,L2)]`

c) `{k:v for k,v in zip(k, v)}`

ENUMERATE

- enumerate produces tuples (index, element)
- index starts at 0

```
>>> S = 'spam'
>>> for (index, item) in enumerate(S):
        print(item, 'appears at offset', index)


s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

AGENDA

- any & all
- zip & enumerate
- **higher-order functions**
- classes and OOP
- exceptions
- file access
- working with CSV and JSON
- coding style

FUNCTIONS AS 'FIRST-CLASS CITIZENS'

- first-class means:
  1. you can passing functions to other functions
  2. you can return functions from other functions
  3. you can assigning functions to variables
  4. you can store functions in data structures (e.g. a list of functions)

- examples : map(f, list), sorted(list, key=f), max(list, key=f) where f is a function

WHAT WILL BE PRINTED?

```
1 def test():
2     print ("test")
3
4 def invoker(f):
5     f()
6
7 invoker(test)
```

```
1 def echo(message):
2     print(message)
3
4 def invoker(f, arg):
5     f(arg)
6
7 invoker(echo, 'Hi')
```

WHAT WILL BE PRINTED?

- calling a function vs. passing a function

```
1 def test():
2     print ("test")
3
4 def invoker(f):
5     f()
6
7 invoker(test)
```

```
1 def test():
2     print ("test")
3
4 def invoker(f):
5     pass
6
7 invoker(test())
```

THE FUNCTION DECORATOR @

```
1  import time
2
3  def timing_function(func):
4      def wrapper():
5          t1 = time.time()
6          func() # call func()
7          t2 = time.time()
8          print('Time it took to run function: ' + str(t2 - t1))
9      return wrapper
10
11 @timing_function
12 def func():
13     time.sleep(2)
14
15 # will actually call wrapper
16 func()
```

```
Time it took to run function: 2.000256299972534
```

MAP

- map : apply a function to (every item of) an iterable and return a list of the results
- note: this can also be done with list comprehension or for-loop

```
>>> def square(n):
        return n*n

>>> lis = list(range(5))
>>> lis
[0, 1, 2, 3, 4]
>>> map(square, lis)
<map object at 0x02F78C70>
>>> list(map(square, lis))
[0, 1, 4, 9, 16]
>>> [x*x for x in lis]
[0, 1, 4, 9, 16]
```

AGENDA

- any & all
- zip & enumerate
- higher-order functions
- **classes and OOP**
- exceptions
- file access
- working with CSV and JSON
- coding style

PYTHON VS. JAVA

- class creates a new class object and assigns it to a name
- no declaration of class or instance variables
- no new operator in Python
- class or instance variables can be created outside a class (is a bad idea)

- OOP in Python is very elaborate, e.g. multiple inheritance

PYTHON VS. JAVA

- multiple classes can be defined within modules, just like functions
- constructor method is like any other method, but with a special name: `__init__`
- 'self' in Python = 'this' in Java
- all instance methods have 'self' as first argument

CLASS AND INSTANCE VARIABLES

- an instance variable can be different for each instance
  - we must first create an instance
  - are (preferably) created in the constructor `__init__`
  - self.x = ... creates an instance variable x

- a class variable is the same for all instances
  - same as static variable in Java
  - are created by assignments outside the constructor in the class
  - for storing constants, setting default values, or tracking data across all instances

CLASS AND INSTANCE VARIABLES

```
 1  class MyClass:
 2      data = 'spam' # class var
 3      def __init__(self, value):
 4          self.data = value  # instance var (instance.data)
 5      def display(self):
 6          print(self.data, MyClass.data) # instance var, class var
 7
 8  y1 = MyClass(11)
 9  y2 = MyClass(22)
10  y3 = MyClass(33)
11  y1.display()
12  y2.display()
13  y3.display()
14
15  # change class var, bad idea, but possible
16  MyClass.data = 'SPAM'
17  y1.display()
18  y2.display()
19  y3.display()
```

```
11 spam
22 spam
33 spam
11 SPAM
22 SPAM
33 SPAM
```

CLASS AND INSTANCE METHODS

- instance method can only be called after creating an instance
- class method is available in the class, no need to first create an instance
  - same as static method in Java, e.g. **`java.lang.Math.sin()`** can be called without creating an instance

INSTANCE METHODS

- first parameter in def is 'self', self refers to the instance of the class
- but you should not include 'self' when you call setname() or display()

```
1  class Student:
2      def __init__(self, name):
3          self.name = name
4      def display(self):
5          print(self.name)
6
7  x = Student('King Arthur')
8  x.display()
9
10 y = Student('Robin Hood')
11 y.display()
```

```
King Arthur
Robin Hood
```

WHAT WILL BE PRINTED?

```
def show_type():
    print('global function')

class MyClass:
    def __init__(self):
        self.show_type()

    def show_type(self):
        print('instance method')

t = MyClass()
```

```
def show_type():
    print('global function')

class MyClass:
    def __init__(self):
        show_type()

    def show_type(self):
        print('instance method')

t = MyClass()
```

CLASS METHODS

- class method is available in all subclasses
- class method receives the class itself instead of an instance
- use @classmethod decorator
- class method can access class attributes and other methods

- why not use a global function ? because it logically belongs to the class

CLASS METHODS

```
1  class  MyClass:
2      numInstances = 0
3
4      def __init__(self):
5          # update class variable
6          MyClass.numInstances = MyClass.numInstances + 1
7          print(MyClass.numInstances)
8
9      @classmethod
10     def howManyInstances(cls):
11         # cls refers to the class itself
12         print('# of instances created: ', cls.numInstances)
13
14 a = MyClass()
15 b = MyClass()
16 c = MyClass()
17 MyClass.howManyInstances() # call from class
18 a.howManyInstances() # call from instance
```

```
1
2
3
# of instances created:  3
# of instances created:  3
```

MODULES VS. CLASSES

- they both (usually) contain variables and functions
- they both have there own namespace
- a class is a blueprint for an object; you can create multiple instances of a class (with different initialization)
- with inheritance you can reuse code in subclasses

- use modules to collect functionality into logical units
- use classes as blueprints for objects if that seems a good model / abstraction for your problem

WHAT WILL BE PRINTED?

```
1  x = 11 # global (module) var
2
3  def f():
4      x = 22 # local (function) var
5      print(x)
6
7  class C:
8      x = 33 # class var
9
10     def __init__(self, value):
11         self.x = value # make instance var in constructor
12
13     def m(self, value):
14         return self.x + value
15
16 print(x)
17 f()
18
19 print(C.x)
20
21 obj = C(5) # make instance
22 print(obj.x)
23 print(obj.m(4)) # call instance method
```

```
11
22
33
5
9
```

PRIVATE ATTRIBUTES

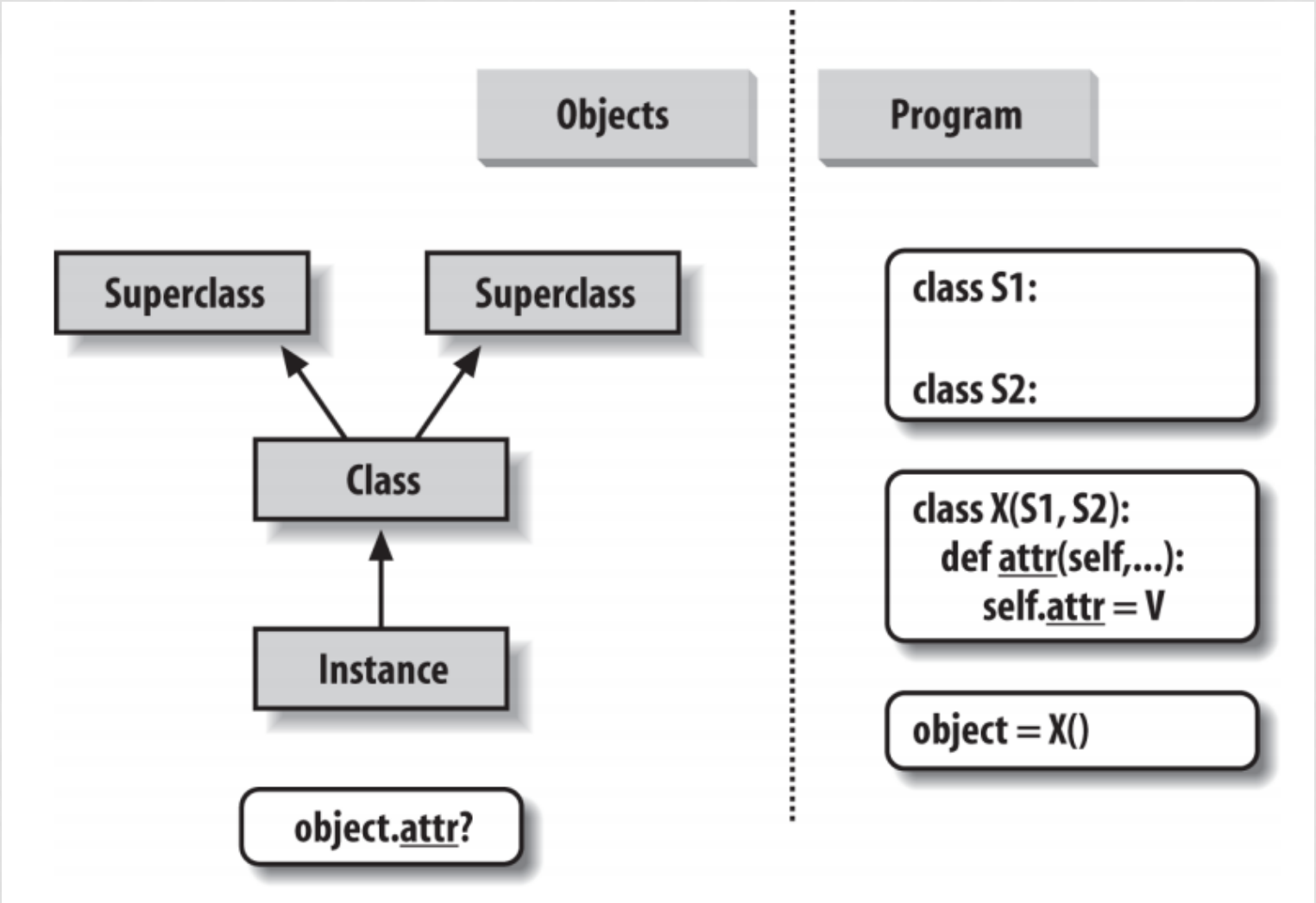
```
1  class Circle:
2      # Construct a circle object
3      def __init__(self, radius = 1):
4          self.__radius = radius
5
6      def getRadius(self):
7          return self.__radius
8
9  c= Circle(5)
10 print(c.getRadius())
11 print(c.__radius)
```

```
5
Traceback (most recent call last):
  File "private_att.py", line 11, in <module>
    print(c.__radius)
AttributeError: 'Circle' object has no attribute '__radius'
```

INHERITED FROM CLASS OBJECT

- every class has at least one superclass called object
  - class ClassName is same as class ClassName(object)
- inherited from object :
  - constructor : `__init__`
  - printing : `__str__`
  - compare: `__eq__(other)`
  - dictionary containing the class's namespace: `__dict__`
  - and many more

POLYMORPHISM : OVERRIDING METHODS



POLYMORPHISM : OVERRIDING METHODS

- program creates a tree of objects in memory to be searched by attribute inheritance
- each reference to an object attribute creates a new bottom-up tree search (same as with Java)

OVERRIDING METHODS

```
 1  class MySuper:
 2      def __init__(self, value=0):
 3          self.value=value
 4      def display(self):
 5          print("Supervalue = ", self.value)
 6
 7  # MyClass is-a MySuper
 8  class MyClass(MySuper):
 9      # explicit reference to super class constructor
10      def __init__(self, value):
11          super().__init__(value)
12      # override display() in MySuper
13      def display(self):
14          print("Subvalue = ", self.value)
15
16  x = MySuper(99)
17  y = MyClass(88)
18  x.display()
19  y.display()
```

```
Supervalue =  99
Subvalue =  88
```

```
 1  class Person:
 2      def __init__(self, name, job=None, pay=0):
 3          self.name = name
 4          self.job = job
 5          self.pay = pay
 6
 7      def give_raise(self, percent):
 8          self.pay = int(self.pay * (1 + percent))
 9
10      def __str__(self):
11          return '[Person: %s, %s]' % (self.name, self.pay)
12
13  class Manager(Person):
14      def __init__(self, name, pay):
15          # reuse superclass
16          super().__init__(name, 'mgr', pay)
17      def give_raise(self, percent, bonus=.10):
18          Person.give_raise(self, percent + bonus)
19
20  if __name__ == '__main__':
21      bob = Person('Bob Smith', job='dev', pay=100)
22      sue = Person('Sue Jones', job='dev', pay=200)
23      tom = Manager('Tom Jones', 300)
24
25      for obj in (bob, sue, tom):
26          obj.give_raise(.10)
27          print(obj)
```

```
[Person: Bob Smith, 110]
[Person: Sue Jones, 220]
[Person: Tom Jones, 360]
```

ABSTRACT SUPERCLASS

- parts of its behavior must be provided by subclass
- you cannot instantiate a class with abstract methods
- abstract superclass inherits from ABC (Abstract Base Class)

```
1  from abc import ABC, abstractmethod
2
3  class Super(ABC):
4      @abstractmethod
5      def action(self):
6          pass
7
8  class Sub(Super):
9      def action(self):
10         print('spam')
11
12 x = Sub()
13 x.action()
```

AGENDA

- any & all
- zip & enumerate
- higher-order functions
- classes and OOP
- **exceptions**
- file access
- working with CSV and JSON
- coding style

EXCEPTIONS

- runtime errors are thrown as exceptions
- an exception is an **object** that represents an error that prevents execution from proceeding normally
- how can a method notify its caller an exception has occurred ?
- goal : separating the detection of an error from the handling of an error

EXCEPTIONS

- try/except/else/finally
  - try : block of code
  - except : catch and handle exceptions
  - else : only if no exception applies
  - finally : *always* do this ('cleanup actions')
- raise
  - raise an exception
- Java: Exceptions which are not RuntimeExceptions *must* be handled (the compiler forces you to check and handle these)

TRY/EXCEPT/ELSE/FINALLY

```
 1  try:
 2      <body>
 3  except <ExceptionType1>:
 4      <handler1>
 5  ...
 6  except <ExceptionTypeN>:
 7      <handlerN>
 8  except:
 9      <handlerExcept>
10  else:
11      <process_else>
12  finally:
13      <process_finally>
```

TRY/EXCEPT/ELSE/FINALLY

```
 1  def fetch(s, index):
 2      return s[index]
 3
 4  try:
 5      fetch('spam', 4)
 6  except IndexError:
 7      print('Index out of range')
 8  finally:
 9      print('This is done always !')
10
11  print('Continuing...\n')
```

```
Index out of range
This is done always !
Continuing...
```

```
 1  def test(x, y):
 2      result = 0
 3      try:
 4          result = x / y
 5      except ZeroDivisionError:
 6          print('Division by zero!')
 7      except SyntaxError:
 8          print('A comma may be missing in the input')
 9      except Exception:
10          print('Something wrong in the input')
11      else:
12          print('No exceptions')
13      finally:
14          print('The finally clause is executed')
15      return result
16
17  r = test(10, 5)
18  r = test(4, 0)
```

```
No exceptions
The finally clause is executed
Division by zero!
The finally clause is executed
```

PYTHON BUILT-IN EXCEPTIONS

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

ASSERT

- if possible, apply TDD : define assertions before implementing a function
- assert statement syntax: `assert test [, message]`
- this works as:

```
if not test raise AssertionError(message)
```

```
1  def hex_to_bin(hex_number):
2      return bin(hex_number)
3
4  assert hex_to_bin(0x0) == '0b0'
5  assert hex_to_bin(0x12EF) == '0b1001011101111'
6  assert hex_to_bin(-0x12EF) == '-0b1001011101111'
7  assert hex_to_bin(0xABC123EF) == '0b10101011110000010010001111101111'
```

AGENDA

- any & all
- zip & enumerate
- higher-order functions
- classes and OOP
- exceptions
- **file access**
- working with CSV and JSON
- coding style

CONTEXT MANAGEMENT : WITH/ AS

```
with expression [as variable]:
    with-block
```

- with/as statement can be an alternative for try/finally
- to guarantee that cleanup-actions are done regardless of any exceptions that may occur in the block

```
with open(filename, 'w') as myfile:
    ...process myfile, auto-closed on statement exit...
```

FILE ITERATOR

```
1  with open('input.txt', 'r') as f:
2      for line in f:
3          print (line)
4
5  with open('output.txt', 'w') as f:
6      f.write('Hi there!')
```

- if you open a file using with-as, it will be closed automatically

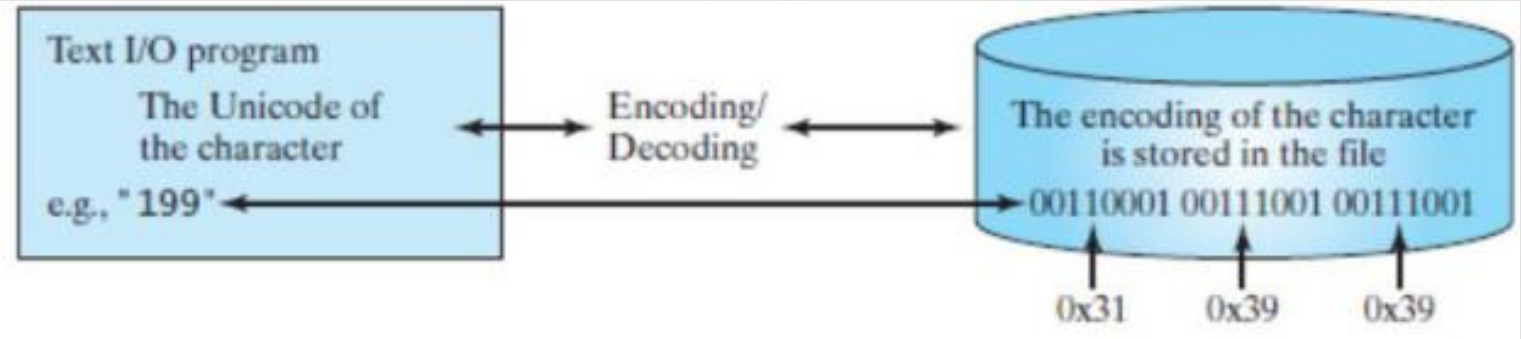
OPEN MODES

- open mode distinguishes between text and binary
  - open modes : r, r+, w, w+, a, a+
  - text is default, add 'b' for binary file (rb, wb)

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of the file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open a disk file for updating (reading and writing)

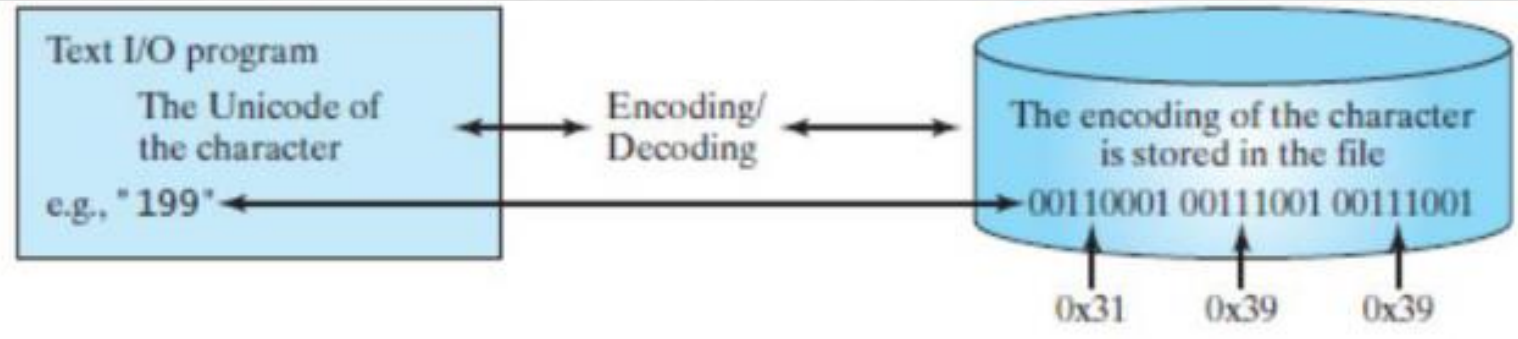
TEXT AND BINARY FILES

- bytes are binary, characters are an abstraction from bytes
- binary files contain only raw bytes
- text files contain Unicode characters
  - Python decodes content from Unicode on input and encodes to Unicode on output
  - one Unicode character can be represented by several bytes (1, 2, or 4 bytes per character)



ENCODING/DECODING

- a string is a sequence of Unicode characters
- when a file is opened in text mode (default), reading its data automatically decodes its content and returns it as a str
- writing takes a str and automatically encodes it as bytes



```python
city = 'São Paolo'
b = city.encode('utf_8')
print(1, b)
print(2, b.decode('utf_8'))

fp = open('city.txt', 'w')
fp.write('São Paolo')
fp.close
fp = open('city.txt', 'r')
print(3, fp.read())
fp.close
fp = open('city.txt', 'rb')
print(4, fp.read())
fp.close
fp = open('city.txt', 'r', encoding='latin-1')
print(5, fp.read())
fp.close
fp = open('city.txt', 'r', encoding='utf_8')
print(6, fp.read())
fp.close
```

```
1  b'S\xc3\xa3o Paolo'
2  São Paolo
3  São Paolo
4  b'S\xe3o Paolo'
5  São Paolo
Traceback (most recent call last):
  File "test_read_and_write.py", line 19, in <module>
    print('6 ',fp.readline())
  File "D:\Python\lib\codecs.py", line 321, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe3 in position 1: invalid continuation byte
```

READLINES

- read() and readlines() load entire file into memory
- readline() and read(n) read only part of a file

```
C:\temp> python
>>> file = open('data.txt')                    # open input file object: 'r' default
>>> lines = file.readlines()                   # read into line string list
>>> for line in lines:                         # BUT use file line iterator! (ahead)
...     print(line, end='')                    # lines have a '\n' at end
...
Hello file world!
Bye   file world.
```

```
>>> file = open('data.txt', 'a')              # open in append mode: doesn't erase
>>> file.write('The Life of Brian')           # added at end of existing data
>>> file.close()
>>>
>>> open('data.txt').read()                   # open and read entire file
'Hello file world!\nBye   file world.\nThe Life of Brian'
```

EOL CHARACTER

- end-of-line characters
  - Windows '\r\n\'
  - Unix '\n'
- on Windows : when reading, '\r\n' is translated to '\n', when writing the reverse

AGENDA

- any & all
- zip & enumerate
- higher-order functions
- classes and OOP
- exceptions
- file access
- **working with CSV and JSON**
- coding style

CSV EXAMPLE

```
1  id,name,age,height,weight
2  1,Alice,20,62,120.6
3  2,Freddie,21,74,190.6
4  3,Bob,17,68,120.0
```

```
1  import csv
2  max_age = 0
3  with open('people.csv') as f:
4      reader = csv.DictReader(f)
5      for row in reader:
6          # process row
7          age = int(row["age"])
8          print(age)
9          if age > max_age:
10             max_age = age
11             oldest_person = row["name"]
12 if max_age > 0:
13     print ("The oldest person is %s, who is %d years old." %
14         (oldest_person, max_age))
```

```
20
21
17
The oldest person is Freddie, who is 21 years old.
```

CSV.DICTREADER()

- creates an object that operates like a regular reader but maps the information in each row to an OrderedDict whose keys are given by the optional fieldnames parameter
- if fieldnames is omitted, the values in the first row of file f will be used as the fieldnames

class `csv.`**`DictReader`**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

Create an object that operates like a regular reader but maps the information in each row to an `OrderedDict` whose keys are given by the optional *fieldnames* parameter.

The *fieldnames* parameter is a sequence. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the fieldnames. Regardless of how the fieldnames are determined, the ordered dictionary preserves their original ordering.

WAT IS SERIALIZATION?

- serialization is the process of translating data structures or object state into a format that can be stored (in a file) or transmitted (across a network) and reconstructed later (possibly in a different computer environment)
- when the resulting series of bits is reread according to the serialization format, it can be used to create a clone of the original object
- the opposite operation, extracting a data structure from a series of bytes (raw or text), is called deserialization

SERIALIZATION FORMATS

- wiki : comparison of data serialization formats

Name	Creator-maintainer	Based on	Standardized?	Specification	Binary?	Human-readable?	Supports references?[e]
HOCON	Typesafe Inc.	JSON	No	Optimized Config Object Notation)"	No	Yes	Yes
Ion	Amazon	JSON	No	The Amazon Ion Specification	Yes	Yes	No
JSON	Douglas Crockford	JavaScript syntax	Yes	RFC 7159 (ancillary: RFC 6901, RFC 6902)	No, but see BSON, Smile, UBJSON	Yes	Yes (JSON Pointer (RFC 6901); alternately: JSONPath, JPath, JSPON, json:select(), JSON-LD
KMIP	OASIS	n/a	Yes	Oasis	Yes (Tag, Type, Length, Value)	Yes	No
MessagePack	Sadayuki Furuhashi	JSON (loosely)	Yes	MessagePack format specification	Yes	No	No
Netstrings	Dan Bernstein	N/A	Yes	netstrings.txt	Yes	Yes	No

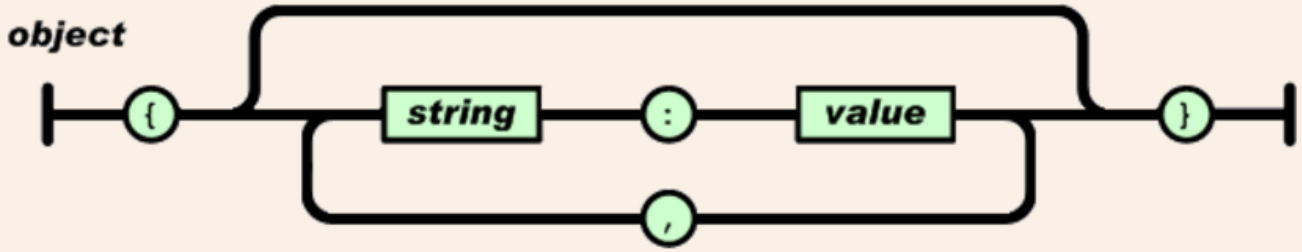
JSON

- representing documents as **k,v pairs**
- for interchanging data (light alternative for XML)
- JavaScript Object Notation came from JavaScript, but is a language-independent data format
- documents may contain other documents, denoted by { }
- only number, string, Boolean, array and documents or k,v pairs (called objects)

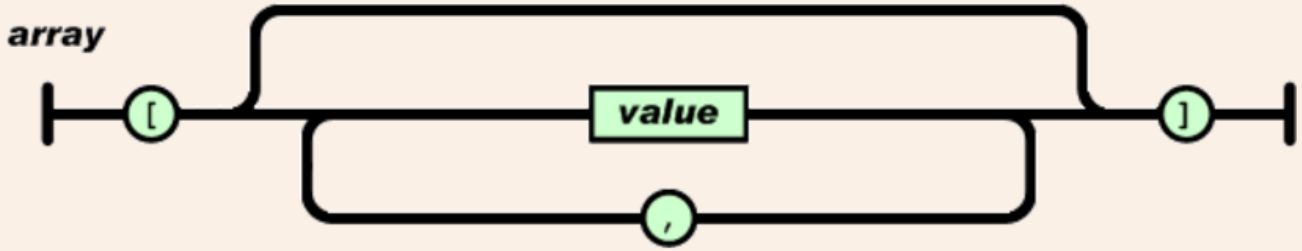
```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

JSON.ORG

An *object* is an unordered set of name/value pairs. An object begins with **{** (left brace) and ends with **}** (right brace). Each name is followed by **:** (colon) and the name/value pairs are separated by **,** (comma).



An *array* is an ordered collection of values. An array begins with **[** (left bracket) and ends with **]** (right bracket). Values are separated by **,** (comma).



JSON - PYTHON MAPPING TYPES

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

```
{
  "headline" : "Apple Reported Fourth Quarter Revenue Today",
  "date" : "2015-10-27T22:35:21.908Z",
  "views" : 1132,
  "author" : {
    "name" : "Bob Walker",
    "title" : "Lead Business Editor"
  },
  "published" : true,
  "tags" : [
    "AAPL",
    { "name" : "city", "value" : "Cupertino" },
    [ "Electronics", "Computers" ]
  ]
}
```

how many:

- keys (" " before :)
- strings (" " after :)
- numbers
- booleans
- arrays [ ]
- objects { } ?

what types are in the array tags-values ?

19.2. `json` — JSON encoder and decoder

- dumps: serialize a Python object to a JSON document (string) using the conversion table
- loads: de-serialize a string containing a JSON document to a Python object using the conversion table

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

JSON EXAMPLE

```
 1  import json
 2  from pprint import pprint
 3
 4  data = {
 5      'name' : 'ACME',
 6      'shares' : 100,
 7      'price' : 542.23
 8  }
 9
10  json_str = json.dumps(data)
11  print(json_str)
12  data = json.loads(json_str)
13  pprint(data)
```

```
> python try_json_1.py
{"price": 542.23, "shares": 100, "name": "ACME"}
{'name': 'ACME', 'price': 542.23, 'shares': 100}
```

```python
from urllib.request import urlopen
from pprint import pprint
import json

url = ('http://maps.googleapis.com/maps/api/geocode/json?'
       'address=Groningen,Hanzeplein')
google_output = urlopen(url).read()
json_output = json.loads(google_output.decode())
pprint (json_output)
```

```
{'results': [{'address_components': [{'long_name': 'Hanzeplein',
                                       'short_name': 'Hanzeplein',
                                       'types': ['route']},
                                      {'long_name': 'Oosterparkwijk',
                                       'short_name': 'Oosterparkwijk',
                                       'types': ['sublocality_level_1',
                                                 'sublocality',
                                                 'political']},
                                      {'long_name': 'Groningen',
                                       'short_name': 'Groningen',
                                       'types': ['locality', 'political']},
                                      {'long_name': 'Groningen',
                                       'short_name': 'Groningen',
                                       'types': ['administrative_area_level_2',
                                                 'political']},
                                      {'long_name': 'Groningen',
                                       'short_name': 'GR',
                                       'types': ['administrative_area_level_1',
                                                 'political']},
                                      {'long_name': 'Netherlands',
                                       'short_name': 'NL',
                                       'types': ['country', 'political']}
```

AGENDA

- any & all
- zip & enumerate
- higher-order functions
- classes and OOP
- exceptions
- file access
- working with CSV and JSON
- **coding style**

CODING STYLE

- PEP-8 : www.python.org/dev/peps/pep-0008
  - PEP : Python Enhancement Proposal
- Google : google.github.io/styleguide/pyguide.html

- `$ pip install pep8`
- `pep8 --show-source myfile.py`

CODING STYLE

- be consistent
- avoid global variables
- use () sparingly

```
while x:
    x = bar()
if x and y:
    bar()
if not x:
    bar()
return foo
for (x, y) in d.items():
    pass
```

- naming
  - `joined_lower` for everything
  - except camelcase for:

    `ClassName, ExceptionName, GLOBAL_CONSTANT_NAME`

- indent with 4 spaces (no hard tabs)

CODING STYLE

- one statement per line
- example line continuation

```
def __init__(self, first, second, third,
             fourth, fifth, sixth):
    output = (first + second + third
              + fourth + fifth + sixth)
```

- whitespace

```
def make_squares(key, value=0):
    """Return a dictionary and a list..."""
    d = {key: value}
    l = [key, value]
    return d, l
```

```
if x == 4:
    print(x, y)
x, y = y, x
```

CODING STYLE

- unless obvious, docstring of function should describe arguments, return values and exceptions raised
- comments : why (rationale) & how code works

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:        # true iff i is a power of 2
```

- modules should be importable
  - always check '__main__'

```
if __name__ == '__main__':
    main()
```