CSIT121
Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 9 outline

- The complexities of strings, bytes, and byte arrays
- The ins and outs of string formatting
- The mysterious regular expression
- Serializing objects

Strings

- Strings are a basic primitive type in Python. We've used them in almost every example.
- Strings are an immutable sequence of characters.
- Python strings are all represented in Unicode characters. So strings can be used to represent accented characters, Chinese, Greek, Cyrillic or Farsi characters.

String manipulation

- In Python, strings are created by wrapping a sequence of characters in single or double quotes.
- Multiline strings can be created using three quote characters.
- Multiple hardcoded strings can be concatenated together by placing them side by side or using the ‘+’ operator.

```
a = "hello"
b = 'world'
c = '''a multiple
line string'''
d = """more multiple
line string"""
e = ("Three " "Strings " "Together")
f = "More " + "Strings " + "Together"
print(e)
Three Strings Together
print(f)
More Strings Together
print(c)
a multiple
line string
```

String manipulation

- Strings can also be iterated over (character by character), indexed, sliced. The syntax is the same as for lists.
- The ‘str’ class has numerous methods to make manipulating strings easier.
- We will have a look some common ones.
- You can always use the ‘dir’ and ‘help’ commands in the Python interpreter to check those str methods.

```
dir(str)
['__add__', '__class__', '__contains__', '__delattr__
', '__dir__', '__doc__', '__eq__', '__format__', '__g
e__', '__getattribute__', '__getitem__', '__getnewarg
s__', '__getstate__', '__gt__', '__hash__', '__init__
', '__init_subclass__', '__iter__', '__le__', '__len_
_', '__lt__', '__mod__', '__mul__', '__ne__', '__new_
_', '__reduce__', '__reduce_ex__', '__repr__', '__rmo
d__', '__rmul__', '__setattr__', '__sizeof__', '__str
__', '__subclasshook__', 'capitalize', 'casefold', 'c
enter', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', '
isalpha', 'isascii', 'isdecimal', 'isdigit', 'isident
ifier', 'islower', 'isnumeric', 'isprintable', 'isspa
ce', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'removeprefix', '
removesuffix', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlin
es', 'startswith', 'strip', 'swapcase', 'title', 'tra
nslate', 'upper', 'zfill']
help(str.strip)
Help on method_descriptor:

strip(self, chars=None, /)
    Return a copy of the string with leading and trai
ling whitespace removed.

    If chars is given and not None, remove characters
 in chars instead.
```

String manipulation

- ‘isalpha’, ‘isupper/islower’, ‘startswith/endswith’, ‘isspace’ methods return Boolean values to indicate whether the characters in a string match a certain pattern.
- ‘istitle’ returns True if the first character of each word is capitalized and all other characters are lowercase.
- ‘isdigit’, ‘isdecimal’, ‘isnumberic’ methods check the numberic value containing by a string. However, the period character (.) is not considered as a decimal character, so ‘45.2’.isdecimal() returns False.
- The real decimal character is represented by Unicode value 0660, i.e., 45.2 (or 45\u06602).

String manipulation

Some pattern-matching methods do not return Booleans.

- “count()” method tells us how many times a given substring shows up in the string.
- ”find()”, “index()”, “rfind()”, “rindex” methods tell us the potion of a given substring within the original string.
- “find()” methods return -1 if the substring can’t be found, while ‘index()’ raises ValueError.
- “rfind()” and “rindex()” methods start searching from the end of the string.

```
s = "hello world"
s.count('l')
3
s.find('1')
-1
s.find('l')
2
s.index('l')
2
s.index('1')
Traceback (most recent call last):
  File "<pyshell#283>", line 1, in <module>
    s.index('1')
ValueError: substring not found
s.rfind('l')

9
s.rindex('l')

9
```

String manipulation

Most of the remaining string methods return transformations of the string, and the input strings remains unmodified, i.e., a brand new string is returned instead.

- ‘upper()’, ‘lower()’, ‘capitalize()’, ‘title()’ methods create new strings with all alphabetical characters in the given format.
- ‘translate()’ method use a dictionary to map arbitrary input characters to specified output characters.
- ‘split()’ and ‘rsplit()’ methods accept a substring and splits the string into a list of strings wherever that substring occurs. You can also limit the number of resultant strings with a second parameter.
- ‘partition()’ and ‘rpartition()’ methods split the string at only the first or the last occurrence of the substring, and return a tuple of three values: characters before the substring, the substring itself, and the characters after the substring.
- ‘join()’ method accepts a list of strings, and returns a single string combing all strings together.
- ’replace()’ method accepts two arguments and returns a string where each instance of the first argument has been replaced with second argument.

String manipulation

```
>>> s = 'hello world, how are you'
>>> s2 = s.split(' ')
>>> s2
['hello', 'world,', 'how', 'are', 'you']
>>> '#'.join(s2)
'hello#world,#how#are#you'
>>> s.replace(' ', '**')
'hello**world,**how**are**you'
>>> s.partition(' ')
('hello', ' ', 'world, how are you')
>>> s.upper()
'HELLO WORLD, HOW ARE YOU'
>>> s.title()
'Hello World, How Are You'
>>> s.capitalize()
'Hello world, how are you'
>>> s.lower()
'hello world, how are you'
```

String formatting

- Python3 has powerful string formatting and templating mechanisms that allow us to construct strings comprised of hardcoded text and interspersed variables.
- A string can be turned into a format string (called an f-string) by prefixing the opening quotation mark with an f, such as f”hello world”. Then the value of variables from the surrounding scope can be replaced the variables between the curly bracket ({ }) in the string.

```
>>> name = "Dusty"
>>> activity = "writing"
>>> formatted = f"Hello {name}, you are currently {activity}."
>>> print(formatted)
Hello Dusty, you are currently writing.
>>> name = "World"
>>> print(formatted)
Hello Dusty, you are currently writing.
```

Escaping braces

- If we want to display the braces in a string, we need a double brace, i.e., {{ or }}. The following example using Python to format a Java program.

```
>>> classname="MyClass"
>>> python_code="print('hello world')"
>>> template=f"""
... public class {classname} {{
...     public static void main(String[] args)){{
...         System.out.println("{python_code}");
...     }}
... }}"""
>>> print(template)

public class MyClass {
    public static void main(String[] args)){
        System.out.println("print('hello world')");
    }
}
```

f-strings with Python code

- Besides the values in primitives, we can also pass complex objects, including lists, tuples, dictionaries, and arbitrary objects, to a string with the f-string.
- We can even do multiple levels of lookup if we have nested data structures.

Using a tuple and a dictionary

```
>>> emails = ("a@example.com", "b@example.com")
>>> message = {
...     "subject": "You Have Mail!",
...     "message": "Here's some mail for you!",
... }
>>> formatted = f"""
... From: <{emails[0]}>
... To: <{emails[1]}>
... Subject: {message['subject']}
... {message['message']}"""
>>> print(formatted)

From: <a@example.com>
To: <b@example.com>
Subject: You Have Mail!
Here's some mail for you!
```

Using a nested data structure

```
>>> message["emails"] = emails
>>> formatted = f"""
... From: <{message['emails'][0]}>
... To: <{message['emails'][1]}>
... Subject: {message['subject']}
... {message['message']}"""
>>> print(formatted)

From: <a@example.com>
To: <b@example.com>
Subject: You Have Mail!
Here's some mail for you!
```

f-strings with Python code

- However, the best option is to have an Email class.

```
class EMail:
    def __init__(self, from_addr, to_addr, subject, message):
        self.from_addr = from_addr
        self.to_addr = to_addr
        self.subject = subject
        self._message = message

    def message(self):
        return self._message
```

```
>>> email = EMail(
...     "a@example.com",
...     "b@example.com",
...     "You Have Mail!",
...     "Here's some mail for you!",
... )
>>> formatted = f"""
... From: <{email.from_addr}>
... To: <{email.to_addr}>
... Subject: {email.subject}
...
... {email.message()}"""
>>> print(formatted)

    From: <a@example.com>
    To: <b@example.com>
    Subject: You Have Mail!

    Here's some mail for you!
```

f-strings with Python code

- Alternative, you can create a formatted() method in the Email class.

```
class EMail:
    def __init__(self, from_addr, to_addr, subject, message):
        self.from_addr = from_addr
        self.to_addr = to_addr
        self.subject = subject
        self._message = message

    def message(self):
        return self._message

    def formatted(self):
        return f"""
            From: <{self.from_addr}>
            To: <{self.to_addr}>
            Subject: {self.subject}

            {self._message}"""
```

```
>>> email = EMail(
...     "a@example.com",
...     "b@example.com",
...     "You Have Mail!",
...     "Here's some mail for you!",
... )
>>> print(email.formatted())

            From: <a@example.com>
            To: <b@example.com>
            Subject: You Have Mail!

            Here's some mail for you!
```

Making it look right

- It is nice to be able to include variables in template strings, but sometimes the variables need a bit coercion to make them look better. The following example performs some calculations with currency, we end up with a long decimal which we don't want to show up in our template.

```
>>> subtotal = 12.32
>>> tax = subtotal * 0.07
>>> total = subtotal + tax
>>> print(f"Sub: ${subtotal} Tax: ${tax} Total: ${total}")
Sub: $12.32 Tax: $0.8624 Total: $13.182400000000001
```

Making it look right

- To fix it, we can use the format() method by including some information inside the curly braces to adjust the formatting of the parameters.

```
>>> subtotal = 12.32
...
>>> tax = subtotal * 0.07
...
>>> total = subtotal + tax
...
>>> print("Sub: ${0:0.2f} Tax: ${1:0.2f} Total: ${total:0.2f}".format(subtotal,tax,total=total))
...
Sub: $12.32 Tax: $0.86 Total: $13.18
```

The number/keyword before the colon is the index/keyword of format() arguments.
0: for values lower than one, make sure a zero is displayed on the left-hand of the decimal point
.: show a decimal point
2: show two places after the decimal
f: format the input value as a float

Making it look right

- We can also specify that each number should take up a particular number of characters on the screen by placing a value before the period.

```
>>> def printOrder(orders):
...     print("PRODUCT    QUANTITY    PRICE    SUBTOTAL")
...     for product, price, quantity in orders:
...         subtotal = price * quantity
...         print(
...             f"{product:10s}{quantity: ^9d}    "
...             f"${price: <8.2f}${subtotal: >7.2f}"
...         )
...
...
>>> orders = [("burger", 2, 5), ("fries", 3.5, 1), ("cola", 1.75, 3)]
>>> printOrder(orders)
    PRODUCT    QUANTITY    PRICE    SUBTOTAL
    burger        5        $2.00    $  10.00
    fries         1        $3.50    $   3.50
    cola          3        $1.75    $   5.25
```

- s means it is a string variable (d for integers and f for floats). For floating-point numbers, the % type will multiply by 100 and format a float as a percentage.
- 10 (9) means it should take up 10 (9) characters.
- ^ tells us that the number should be aligned in the center of this available padding. < and > are the left and right aligns;
- (space) tells the formatter to use a space as the padding character. With integers, instead of spaces, the extra characters are zeros.
- product, quantity, price and subtotal are the variable being formatted.

Regular expression

- Parsing strings to match arbitrary patterns is not an easy job.
- In most programming languages, string-parsing is handled by regular expressions.
- Even though regular expressions are not object-oriented, the Python regular expression library provides a few classes and objects that you can use to construct and run regular expressions.

Regular expression

Regular expressions are used to solve a common problem: Given a string, determine whether that string matches a given pattern and, optionally, collect substrings that contain relevant information. They can be used to answer questions such as the following:

- Is this string a valid URL/email address/phone number/etc?
- What is the date and time of all warning messages in a log file?
- Which users in /etc/passwd are in a given group?
- What username and document were requested by the URL a visitor typed?

Matching patterns

- Regular expressions are a complicated mini-language, and reply on special characters to match unknown strings.
- Let's start with literal characters, such as letters, numbers, and the space character.
- These literal characters basically always match themselves.

```
import re
search_string = "hello world"
pattern = "hello world"
match = re.match(pattern, search_string)
print(match)
<re.Match object; span=(0, 11), match='hello world'>
if match:
    print("regex matches")


regex matches
```

- re refers to the Python standard library module for regular expressions.
- match() function matches the pattern to the beginning of the string.
- If the pattern were 'ello world', no match would be found. However, the pattern 'hello worl' will match.

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "ello world" "hello world"
'hello world' does not match pattern 'ello world'
fren@UW-XY06K990HQ examples % python3 regex_check.py "hello worl" "hello world"
'hello world' matches pattern 'hello worl'
```

Matching patterns

- ^ and $ characters are used to represent the start and end of the string in the pattern

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "^hello world$" "hello world"
'hello world' matches pattern '^hello world$'
fren@UW-XY06K990HQ examples % python3 regex_check.py "^hello world$" "hello worl"
'hello worl' does not match pattern '^hello world$'
```

- The period character (.) can match any single character.

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "hel.o world" "hello world"
'hello world' matches pattern 'hel.o world'
fren@UW-XY06K990HQ examples % python3 regex_check.py "hel.o world" "helpo world"
'helpo world' matches pattern 'hel.o world'
fren@UW-XY06K990HQ examples % python3 regex_check.py "hel.o world" "helo world"
'helo world' does not match pattern 'hel.o world'
```

Matching patterns

- We can put a set of characters inside square brackets to match any one of those characters.

```
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hel[lp]o world" "hello world"
'hello world' matches pattern 'hel[lp]o world'
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hel[lp]o world" "helpo world"
'helpo world' matches pattern 'hel[lp]o world'
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hel[lp]o world" "helco world"
'helco world' does not match pattern 'hel[lp]o world'
```

- If we want to include a large range of characters inside the square brackets, we can use a dash character (-) to create a range. Any character between the range will match the pattern.

```
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hello [a-z] world" "hello  world"
'hello  world' does not match pattern 'hello [a-z] world'
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hello [a-z] world" "hello b world"
'hello b world' matches pattern 'hello [a-z] world'
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hello [a-zA-Z] world" "hello B world"
'hello B world' matches pattern 'hello [a-zA-Z] world'
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hello [a-zA-Z0-9] world" "hello 2  world"
'hello 2  world' does not match pattern 'hello [a-zA-Z0-9] world'
[fren@UW-XY06K990HQ examples % python3 regex_check.py "hello [a-zA-Z0-9] world" "hello 2 world"
'hello 2 world' matches pattern 'hello [a-zA-Z0-9] world'
```

Escaping characters

- If putting a period character in a pattern matches any arbitrary character, how do we match just a period?
- A generic solution is to use backslashes to escape it.
- Here is a regular expression to match two-digit decimal numbers between 0.00 and 0.99

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "0\.[0-9][0-9]" "0.05"
'0.05' matches pattern '0\.[0-9][0-9]'
fren@UW-XY06K990HQ examples % python3 regex_check.py "0\.[0-9][0-9]" "005"
'005' does not match pattern '0\.[0-9][0-9]'
fren@UW-XY06K990HQ examples % python3 regex_check.py "0\.[0-9][0-9]" "0,05"
'0,05' does not match pattern '0\.[0-9][0-9]'
```

- We can also use the escape symbol followed by a character to represent special characters such as newlines (\n), tabs (\t), whitespaces (\s), letters, numbers, and underscores (\w), and digits (\d).

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "\(abc\]" "(abc]"
'(abc]' matches pattern '\(abc\]'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\s\d\w" " 1a"
' 1a' matches pattern '\s\d\w'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\s\d\w" "\t5n"
'\t5n' does not match pattern '\s\d\w'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\s\d\w" "5n"
'5n' does not match pattern '\s\d\w'
```

Matching multiple characters

- What if we don’t know how many characters to match inside a patter?
- We can use the asterisk (*) character to match a pattern by zero, one or more times.

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "hel*o" "hello"
'hello' matches pattern 'hel*o'
fren@UW-XY06K990HQ examples % python3 regex_check.py "hel*o" "heo"
'heo' matches pattern 'hel*o'
fren@UW-XY06K990HQ examples % python3 regex_check.py "hel*o" "hellllo"
'hellllo' matches pattern 'hel*o'
```

- .*, will match any string.
- [a-z]*, matches any collection of lowercase words, including the empty string.

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "[A-Z][a-z]* [a-z]*\." "A string."
'A string.' matches pattern '[A-Z][a-z]* [a-z]*\.'
fren@UW-XY06K990HQ examples % python3 regex_check.py "[A-Z][a-z]* [a-z]*\." "Apple pie."
'Apple pie.' matches pattern '[A-Z][a-z]* [a-z]*\.'
fren@UW-XY06K990HQ examples % python3 regex_check.py "[A-Z][a-z]* [a-z]*\." "No ."
'No .' matches pattern '[A-Z][a-z]* [a-z]*\.'
fren@UW-XY06K990HQ examples % python3 regex_check.py "[a-z]*.*" ""
'' matches pattern '[a-z]*.*'
```

Matching multiple characters

- The plus (+) sign in a pattern behaves similarly to an asterisk, but it states that the previous pattern must be repeated one or more time.
- The question mark (?) ensures a pattern shows up exactly zero or one times, but not more.

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "\d+\.\d+" "0.4"
'0.4' matches pattern '\d+\.\d+'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\d+\.\d+" "1.002"
'1.002' matches pattern '\d+\.\d+'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\d+\.\d+" "1."
'1.' does not match pattern '\d+\.\d+'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\d?\d%" "1%"
'1%' matches pattern '\d?\d%'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\d?\d%" "99%"
'99%' matches pattern '\d?\d%'
fren@UW-XY06K990HQ examples % python3 regex_check.py "\d?\d%" "999%"
'999%' does not match pattern '\d?\d%'
```

Grouping patterns together

- What if we want a repeating sequence of characters?
- We can enclose any set of patterns in parentheses, allowing them to be treated as a repeatable pattern.

```
fren@UW-XY06K990HQ examples % python3 regex_check.py "abc{3}" "abccc"
'abccc' matches pattern 'abc{3}'
fren@UW-XY06K990HQ examples % python3 regex_check.py "(abc){3}" "abccc"
'abccc' does not match pattern '(abc){3}'
fren@UW-XY06K990HQ examples % python3 regex_check.py "(abc){3}" "abcabcabc"
'abcabcabc' matches pattern '(abc){3}'


fren@UW-XY06K990HQ examples % python3 regex_check.py "[A-Z][a-z]*( [a-z]+)*\." "Eat."
'Eat.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.'
fren@UW-XY06K990HQ examples % python3 regex_check.py "[A-Z][a-z]*( [a-z]+)*\." "Eat more food."
'Eat more food.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.'
fren@UW-XY06K990HQ examples % python3 regex_check.py "[A-Z][a-z]*( [a-z]+)*\." "A good meal."
'A good meal.' matches pattern '[A-Z][a-z]*( [a-z]+)*\.'
```

Getting information from regular expressions

- Python regular expression (re) module provides an object-oriented interface to enter the regular expression engine.
- re.match() function returns None for no matching and a useful object for matching.
- The returned object can help to answer the question: if this string matches this pattern, what is the value of a relevant substring?

```
>>> pattern = "^[a-zA-Z.]+@([a-z.]*\.[a-z]+)$"
>>> search_string = "some.user@example.com"
>>> match = re.match(pattern, search_string)
>>> if match:
...     domain = match.groups()[0]
...     print(domain)
...
...
example.com
```

Getting information from regular expressions

The re module also provides other useful functions.

- The search() function finds the first instance of a matching pattern.
- The findall() function finds all non-overlapping instances of the matching pattern

```
import re
re.findall('a. ', 'abacadefagah')
[]
re.findall('a.', 'abacadefagah')
['ab', 'ac', 'ad', 'ag', 'ah']
re.findall('a(.)', 'abacadefagah')
['b', 'c', 'd', 'g', 'h']
re.findall('(a)(.)', 'abacadefagah')
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'g'), ('a', 'h')]
re.findall('((a)(.))', 'abacadefagah')
[('ab', 'a', 'b'), ('ac', 'a', 'c'), ('ad', 'a', 'd'), ('ag', 'a', 'g'),
('ah', 'a', 'h')]
```

Serializing objects

- Nowadays, we take the ability to write data to file and retrieve it later.
- We usually convert data in a nice object or design pattern in memory into some text or binary format for storage, transfer over the network or remote invocation on a distant server. It is also called object serialisation/deserialization.
- Python pickle module is an object-oriented way to store objects directly in a special storage format.
- It converts an object (and all the objects it holds as attributes) into a sequence of bytes.
- The pickle module comprises four basic functions for storing, loading and manipulating objects.

Serializing objects

- The dump method in the pickle module accepts an object to be written and a file-like object to write the serialized bytes to.
- The file-like object must have a write method to handle a bytes argument (a file opened for text output wouldn’t work).
- The load method in the pickle module reads a serialized object from a file-like object. Again, this object must have the proper file-like read and readline arguments to return bytes.
- The pickle module will load the object from these bytes and the load method will return the fully reconstructed object.

Serializing objects

```
import pickle

some_data = [
    "a list",
    "containing",
    5,
    "values including another list",
    ["inner", "list"],
]

with open("pickled_list", "wb") as file:
    pickle.dump(some_data, file)

with open("pickled_list", "rb") as file:
    loaded_data = pickle.load(file)

print(loaded_data)
assert loaded_data == some_data
```

```
['a list', 'containing', 5, 'values
including another list', ['inner', '
list']]
```

- This code stores the objects in the file and then loads from the same file and print the data values.
- We open the file using a with statement for writing and reading purposes, respectively.
- The assert statement raise an error if the newly loaded object was not equal to the original object. However, they are not the same object.
- The dump() function returns a serialized bytes object, and the load() function returns the restored object.

Serialising JSON object

- JavaScript Object Notation (JSON) is human-readable format for exchanging primitive data through the internet. JSON is a standard format that can be interpreted by a wide array of heterogeneous client systems. Hence, JSON is very useful for transmitting data between completely decoupled systems.
- Because JSON can be easily interpreted by JavaScript engines, it is often used for transmitting data from a web server to a JavaScript-capable web browser.
- If the web application serving the data is written in Python, it needs a way to convert internal data into the JSON format.

Serialising JSON object

- Python has a module called json to complete the job for us.
- The json module provides a similar interface to the pickle module, with dump, load, dumps and loads functions.
- The differences between json module and pickle module are:
  - The output of json module functions is valid JSON notation, rather than a pickled object.
  - The json module functions operate on str objects, rather than bytes.
  - Therefore, when dumping to or loading from a file, we need to create text files rather than binary ones.
- The JSON serialiser can only serialise data in primitive types, strings, lists and dictionaries.
- JSON can’t represent methods, so it is not possible to transmit complete objects to JSON. JSON is just a data notation, but not an object notation. So we just need to serialise the object’s __dict__ attribute.

Serialising JSON object

```
import json
class Contact:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return "{} {}".format(self.first, self.last)


c = Contact("John", "Smith")
json.dumps(c.__dict__)
'{"first": "John", "last": "Smith"}'
```

We can also create a custom encoder if the receiving code want the full_name property to be supplied.

Serialising JSON object

```
class ContactEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Contact):
            return {
                "is_contact": True,
                "first": obj.first,
                "last": obj.last,
                "full": obj.full_name,
            }
        return super().default(obj)
```

```
>>> json.dumps(c, cls=ContactEncoder)
'{"is_contact": true, "first": "John", "last": "Smith",
"full": "John Smith"}'
```

Serialising JSON object

- For decoding, we can write a function that accepts a dictionary and checks the existence of the is_contact variable to decide whether to convert it to a contact.

```
def decode_contact(dic):
    if dic.get("is_contact"):
        return Contact(dic["first"], dic["last"])
    else:
        return dic
```

```
>>> json_data = '{"is_contact": true, "first": "John", "la
    st": "Smith", "full": "John Smith"}'
>>> c = json.loads(json_data, object_hook=decode_contact)
>>> c
    <__main__.Contact object at 0x10caeda90>
>>> c.full_name
```

Suggested reading

Python 3 Object-Oriented Programming

- Chapter 8: Strings and Serialization

Python

- https://www.python.org/
- https://docs.python.org/3/howto/regex.html