Chapter 08 - Strings and Serialization

April 4, 2021

0.1 Strings

- a string is an immutable sequence of chaeracters
- the sequence of characters are Unicode and can represent any language

0.2 String Manipulation

- if you have a really long string, you can just seprate out the quotes

[3]:
```
e = ('http://' 'long_sequence'
             'end')

e
```

[3]:
```
'http://long_sequenceend'
```

common boolean methods: - `isalpha` - `issupper/islower` - `startswith/endswith` - `isspace` - `istitle` - `isdigit/isdecimal/isnumeric`

- do not use Boolean numeric checks because of the inconsistencies, you better off using requlars expressions

common non-boolean methods - `count` - `find` - `index` –rfind–rindex‘

common reansforming methods: - `upper` - `lower` - `capitalize` - `title` - `translate`

- when performing the transformation, it is better to assign the transfromed string to the orgnial string since you do not neet it anymore

[4]:
```
value = 'test'
value = value.upper()
value
```

[4]:
```
'TEST'
```

common string methods that return or operate on lists - `split` - `rsplit` - `partition` - `rpartition` - `split` - `join`

[9]:
```
s = "hello world, how are you"
s_part = s.partition(" ")
s_split = s.split(' ')
```

```
print(f's partition: {s_part}')
print(f's split: {s_split}')
```

```
s partition: ('hello', ' ', 'world, how are you')
s split: ['hello', 'world,', 'how', 'are', 'you']
```

0.3 Escaping Braces

- to escape braces, just use double braces `{{}}`

0.4 Decimal Calculations

- we should never use floating-point numbners in currency calculations
- you should use `decimal.Decimal()` objects instead
- to correct formatting issues with decimals, we can use sting formatting

```
print(
    "Sub: ${0:0.2f} Tax: ${1:0.2f} "
    "Total: ${total:0.2f}".format(subtotal, tax, total=total)
)
```

- the `0.2f` format specifier after the colon basically says the following, from left to right:
  - `0`: for the values lower than one, make sure a zero is dispalyed 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

`{product:10s}`: - `s` means it s a string variable - `10` means it should take up 10 characters

`{quantity: ^9d}` - `d` represents an integer value - `9` tells us the value should take up 9 characters on the screen - `^` tells us that the number should be aligned in the center of this avaialble padding; this makes the column look more professional - `(space)` tells us the formatter to use a space as the padding character

0.5 Custom Formatters

```
import datetime
print("{the_date:%Y-%m-%d %I:%M%p }".format(datetime.datetime.now()))
```

0.6 Strings are Unicode

- if you get a string of bytes from a file or a socket, they wont be in `Unicode`
- they will be the built-in type `bytes`
- if we print a byte object, any bytes that map to `ASCII` representations will be printed as their original character, while non-ASCII bytes are printed as hex codes escaped by the `\x` escape sequence
- since there are many unicodes, bytes must be decoded using the same unicode they were encoded with

0.7 Converting bytes to text

- the `decode()` method on the `byte` class accepts a string for the name of the character
- this can be `ASCII`, `UTF-8`, etc

[13]:
```
characters = b'\x63\x6c\x69\x63\x68\xe9'
print(characters)
print(characters.decode("latin-1"))
```

```
b'clich\xe9'
cliché
```

0.8 Converting Text to Bytes

- we use the `encode` method on the `str` class
- the `encode` method also takes in another paramater that decided what happend if it encounters a wrong encoder

```
characters = "cliché"
print(characters.encode("UTF-8"))
```

```
print("encode('ascii' strategy)")
print(characters.encode("ascii", strategy))
print("")
print("encode('ascii' replace)")
print(characters.encode("ascii", replace))
print("")
print("encode('ascii' ignore)")
print(characters.encode("ascii", ignore))
print("")
print("encode('ascii' xmlcharrefreplace)")
print(characters.encode("ascii", xmlcharrefreplace))
print("")
```

0.9 Mutable Byte Strings

- `bytearray` behaves like a list but it only holds bytes
- the constructor for the class accepts a `bytes` object to initalize
- the `extend` method can be used to append another `bytes` object to the existing array

[22]:
```
b = bytearray(b"abcdefgh")
b[4:6] = b"\x15\xa3"
print(b)
```

```
bytearray(b'abcd\x15\xa3gh')
```

- if we want to manipulate a single element in bytearray, we must pass an integer between `0` and `255` as the value
- the integer represents a specifric `bytes` pattern

- a single byte character can be onverted to an integer using the `ord` function

[23]:
```
b = bytearray(b"abcdef")
b[3] = ord(b"g")
b[4] = 68
print(b)
```

```
bytearray(b'abcgDf')
```

- the bytearray type has methods that allow it to behave like a list
  - meaning we can append integer bytes to it
- but it also has methods such as `count` and `find`
- difference between `str` and `bytearray` is that a bytearrayt can be manipulated

0.10 Regular Expressions

- regular expressions are not OOP but python has a libary
- string parsing is almost always left to `re`

0.10.1 Matching Patterns

[25]:
```
import re

search_string = "hello world"
pattern = "hello world"
match = re.match(pattern, search_string)

if match:
    print("regex matches")
```

```
regex matches
```

0.10.2 Matching a Selection of Characters

- you have to use ‘[a-zA-z0-9]

0.11 Escaping Characters

- the backslash `\` is the escape sequence

0.12 Special Classes

- there are `\n`, `\t`, `\d`, `\w`

0.12.1 Matching multiple characters

- use the wild card `*`
- or you can jjust put a plus sign at the end of everything
  - `[a-z]+`

0.12.2 Grouping Patterns Together

- we can use `{}`
- `abcabcabc` == `(abc){3}`

0.12.3 Getting information from regular expressions

- the `group` method returns a tuple of all the groups matched inside the pattern, which you can index to access a specific value
- there is also the import `findall` function and the `search` method

0.12.4 Making repeated regular expressions efficient

- when we use regular expressions, the engine has to convert the pattern string into an internal structutre that makes searching strings fast
- using `re.comple` method makes it so that the conversion step could be done only once
- it returns an object-oriented version of the regular expression that has been complied down and has the methods we explored such as `match`, `search`, `findall`

0.13 Filesystem Paths

- python uses the `os,path` module
- `os.path` is a pain to use
- so python developers developed something called `pathlib`

[29]:
```
import pathlib

path = (pathlib.Path(".") / "subdir" / " subsubdir" / "file.ext").absolute()
print(path)
```

```
C:\Users\Vicktree\Desktop\notebook Draft\OOP python\subdir\ subsubdir\file.ext
```

the common methods are: - `absolute()`, `parent`, `exists()`, `mkdir()`, `exist_ok`

- you can also get the path to objects like `ZIP` files

```
zipfile.ZipFile(Path('nothing.zip'), 'w').writestr('filename',
'contents')
```

0.14 Serializing Objects

- python `pickle` module is an OOP way to store objects directly in a special storage format
- it converts an object into a sequence of bytes that are stored or transported howerver we see fit
- the `dump` method accepts an object to write and a file-like object to write the searlized bytes to
- the object must have a `write` method and that method must know how to handle `bytes` arguments
- the `load` method does exactly the oppsite

- it reads a serialized object from a file-like object
- the object must have proper `read` and `readline` argument that must return 'bytes
- the `pickle` module will load the object from these bytes and the `load` method will return a fully reconstructed object

[31]:
```
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']]
```

- `dump` and `load` behave like the file opening methods except they return or accept `bytes`
- `dump` returns a seralized object
- the `load` takes bytes and returns a restored obejct

0.15 Customizing Pickels

- basic primitives such as integers, floats, and strings can be pickled
- container objects such as `lists`, `dictionaries` can also be pickled provided the contents fo those containers are also pickable
- objects can also be pickled, provided all of its attributes are also pickable
- usually time dependent stuff is not pickable
- things like socket conenctions, database conenctions, etc.
- when `pickle` tries to serialize an objectm it simply tries to stoe the objects `__dict__` attribute
- the `__dict__` is a dictionary mapping all the attribute names on the object to their values
- if there is a `__getstate__` method, it will try to get that value instead of the `__dict__`
- by using `__getstate__` we dont have to worry about pickling time sensentive data such as the `Timer`
- we will lose our timer with that, so we need a way to get it back, luckely we can use '**setstate**
- `__setstate__` can bewe used to implement customizing unpickling
  - `def __setstate__(self, data): self.__dict__ = data self.schedule()`

[33]:
```
from threading import Timer
import datetime
from urllib.request import urlopen

class UpdatedURL:
```

```
    def __init__(self, url):
        self.url = url
        self.contents = ''
        self.last_updated = None
        self.update()

    def __getstate__(self):
        new_state = self.__dict__.copy()
        if 'timer' in new_state:
            del new_state
        return new_state

    def update(self):
        self.contents = urlopen(self.url).read()
        self.last_updated = datetime.datetime.now()
        self.schedule()

    def schedule(self):
        self.timer = Timer(3600, self.update)
        self.timer.setDaemon(True)
        self.timer.start()
```

0.16 Serializing Web Objects

- it is not a good idea to load pickled objects from an untrusted source
- another problem of pickled programs is that only `Python` can load them
- `JSON` is the human-readable format for exchanging primitive data
- in JSON, only data can be `serialized` and no code can be executed
- thus you do not have to worry about inejcting malicious statements into it

- `json` module is simmilar to the `pickle` module with the exception to the output
- the output is a valid `JSON` instead of a `picked` object
- `json` functions operate on `str` objects and not `bytes`
- thus when dumping or loading form a file, we create text files and not binary ones

- `JSON` cannot represent classes or functions, etc.
- `JSON` is not an object, it is only data; it has no behaviors
- there are ways to use `JSONEncoder` to accpet an object and convert it into a dictionary that `json` can digest
- if it does not know how to process the object, we can just call the `super()` method
- note, you could use `json.dumps(c.__dict__)` but that is crass

[39]:
```
import json

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

```
    @property
    def full_name(self):
        return (f"{self.first} {self.last}")

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)

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


c = Contact("John", "Smith")
# json.dumps(c.__dict__)
json.dumps(c, cls=ContactEncoder)
```

[39]: '{"is_contact": true, "first": "John", "last": "Smith", "full": "John Smith"}'