Advanced Python Labs - Solutions

---

1 - Data Types

1a - Basic data types

Question 1

Read the list of fruits from `files/1a-basic_data_types/fruit.txt` and save it in a dictionary. The key to the dictionary will be the first letter of the fruit's name, and the value will be a list of fruits that begin with that letter.

Print the dictionary.

Note

You can use Python's `pprint` module to print the dictionary. Its `pprint()` (pretty print) function is just what you need to print the fruits' data.

Solution

In [58]:

```
from pprint import pprint

fruits = {}
with open('../files/1a-basic_data_types/fruit.txt') as f:
    for name in f:
        name = name.strip().lower()
        letter = name[0]
        try:
            fruits[letter].append(name)
        except KeyError:
            fruits[letter] = [name]

pprint(fruits)
```

```
{'a': ['apple', 'apricot', 'avocado'],
 'b': ['banana',
       'breadfruit',
       'bilberry',
       'blackberry',
       'blackcurrant',
       'blueberry',
       'boysenberry'],
 'c': ['cantaloupe',
       'currant',
       'cherry',
       'cherimoya',
       'chili',
       'cloudberry',
       'coconut',
       'canary melon',
       'cantaloupe',
       'clementine (type of orange)'],
 'd': ['damson', 'date', 'dragonfruit', 'durian'],
 'e': ['elderberry'],
 'f': ['feijoa', 'fig'],
 'g': ['gooseberry', 'grape', 'grapefruit', 'guava'],
 'h': ['huckleberry', 'honeydew', 'honeydew'],
 'j': ['jackfruit', 'jambul', 'jujube'],
 'k': ['kiwi fruit', 'kumquat'],
 'l': ['legume', 'lemon', 'lime', 'loquat', 'lychee'],
 'm': ['mango', 'melon', 'mandarine'],
 'n': ['nectarine', 'nut'],
 'o': ['orange'],
 'p': ['papaya',
       'passionfruit',
       'peach',
       'pepper',
       'pear',
       'persimmon',
       'physalis',
       'plum/prune (dried plum)',
       'pineapple',
       'pomegranate',
       'pomelo',
       'purple mangosteen'],
 'q': ['quince'],
 'r': ['rock melon', 'raspberry', 'rambutan', 'redcurrant'],
 's': ['salal berry', 'satsuma', 'star fruit', 'strawberry'],
 't': ['tangerine', 'tamarillo', 'tomato'],
 'u': ['ugli fruit'],
 'w': ['watermelon',
       'williams pear or bartlett pear',
       'western raspberry (blackcap)',
       'watermelon-see melon']}
```

1b - Specialised data types

Question 1

Use Counter to count the frequency of each character in the following qoute by Fyodor Dostoevsky:

> Man is fond of counting his troubles, but he does not count his joys. If he counted them up as he ought to, he would see that every lot has enough happiness provided for it.

Solution

In [121]:

```
from collections import Counter

quote = "Man is fond of counting his troubles, \
but he does not count his joys. If he counted them up as he ought to, \
he would see that every lot has enough happiness provided for it."

c = Counter(quote)
print('frequencies:', list(c.items()))
```

```
frequencies: [('M', 1), ('a', 5), ('n', 9), (' ', 34), ('i', 7),
('s', 11), ('f', 4), ('o', 16), ('d', 6), ('c', 3), ('u', 9), ('t',
13), ('g', 3), ('h', 12), ('r', 4), ('b', 2), ('l', 3), ('e', 15),
(',', 2), ('j', 1), ('y', 2), ('.', 2), ('I', 1), ('m', 1), ('p',
4), ('w', 1), ('v', 2)]
```

Question 2

Count how many times each word appears in the Tao te ching (`solutions/2b-specialised_data_structures/tao_te_ching.txt`).

What are the 10 most common words? What are the 10 least common words?

Steps

- Read the file
- Load the words into a counter
- (Bonus) Make sure the words contain only letters to avoid counting problems
- Print the most and least common

Solution

In [60]:

```
from collections import Counter
import re

non_alpha_numeric_pattern = re.compile(r'[^A-Za-z]')

def clean_word(word):
    return non_alpha_numeric_pattern.sub('', word)

words = Counter()
with open('../files/1b-specialised_data_structures/tao_te_ching.txt') as f:
    for line in f:
        if line.startswith('#'):
            continue
        line = line.strip()
        clean_words = []
        for word in line.split():
            clean_words.append(clean_word(word))
        words.update(clean_words)

common_words = []
for item in words.most_common(10):
    common_words.append(item[0])
print('most common words are:\n', common_words)

least_common = []
for item in words.most_common()[-10:]:
    least_common.append(item[0])
print('\nleast common words are:\n', least_common)
```

```
most common words are:
 ['the', 'is', 'to', 'of', 'it', 'and', 'not', 'that', 'be', 'The']

least common words are:
 ['barking', 'die', 'finesounding', 'Finesounding', 'prove', 'prove
s', 'hoard', 'behalf', 'heavens', 'sharpen']
```

Question 3

Use python's `csv` module to digest a csv (Comma Seperated Values) file holding a list of data about people, into usable namedtuples

The output should look this way:

```
Name                           Email
============================================================
James Butt                     jbutt@gmail.com
Josephine Darakjy              josephine_darakjy@darakjy.org
Art Venere                     art@venere.org
Lenna Paprocki                 lpaprocki@hotmail.com
Donette Foller                 donette.foller@cox.net
Simona Morasca                 simona@morasca.com
Mitsue Tollner                 mitsue_tollner@yahoo.com
Leota Dilliard                 leota@hotmail.com
Sage Wieser                    sage_wieser@cox.net
Kris Marrier                   kris@gmail.com
```

Steps

- Open the file `exercises/2b-specialised_data_structures/us-500.csv`
- Call `csv.reader()` to read the file data
- Get the reader to digest the first line and use it to setup the namedtuple's fields
- Read the rest of the csv data into a list of namedtuples
- Iterate over the namedtuples and print the requested data using string's format function

Implementation tips (SPOILER ALERT!)

Getting the first cvs line to define the namedtuple

If you have a cvs reader (`cvs.reader()`) object named `creader`, call

```
headers = next(creader)
```

to get the next parsed line. Use this line to define the namedtuple

Converting cvs records into namedtuples

After you define the namedtuple you need to use it to create a namedtuple from each csv record. The csv's reader returns a list of items, but the new namedtuple class you defined needs the list's contents as seperate arguments for its constructor method. Experiment with prefixing the unpacking operator * to the list. (*Tip* - try it out in IPython first using the `print()` function in Python 3 or in Python 2's `__future__` module)

Solution

In [61]:

```
import csv
from collections import namedtuple

people = []
with open('../files/1b-specialised_data_structures/us-500.csv') as f:
    reader = csv.reader(f)
    Data = namedtuple('Data', next(reader))
    for row in reader:
        record = Data(*row)
        people.append(record)

print('{:<30}{:<30}'.format('Name', 'Email'))
print('='*60)
for record in people[:10]:
    name = '{} {}'.format(record.first_name, record.last_name)
    print('{:<30}{:<30}'.format(name, record.email))
```

```
Name                          Email
============================================================
James Butt                    jbutt@gmail.com
Josephine Darakjy             josephine_darakjy@darakjy.org
Art Venere                    art@venere.org
Lenna Paprocki                lpaprocki@hotmail.com
Donette Foller                donette.foller@cox.net
Simona Morasca                simona@morasca.com
Mitsue Tollner                mitsue_tollner@yahoo.com
Leota Dilliard                leota@hotmail.com
Sage Wieser                   sage_wieser@cox.net
Kris Marrier                  kris@gmail.com
```

Question 4

Revisit question 1 from 1a - Basic data types, and reimplement the code for reading the fruits into a dictionary, using defaultdict.

Note - `pprint` doesn't support defaultdict as nicely as dict

Solution

In [62]:

```
from collections import defaultdict

fruits = defaultdict(list)
with open('../files/1a-basic_data_types/fruit.txt') as f:
    for name in f:
        name = name.strip().lower()
        letter = name[0]
        fruits[letter].append(name)

from pprint import pprint
for letter in fruits:
    print(letter, end=": ")
    pprint(fruits[letter])
```

```
a: ['apple', 'apricot', 'avocado']
b: ['banana',
 'breadfruit',
 'bilberry',
 'blackberry',
 'blackcurrant',
 'blueberry',
 'boysenberry']
c: ['cantaloupe',
 'currant',
 'cherry',
 'cherimoya',
 'chili',
 'cloudberry',
 'coconut',
 'canary melon',
 'cantaloupe',
 'clementine (type of orange)']
d: ['damson', 'date', 'dragonfruit', 'durian']
e: ['elderberry']
f: ['feijoa', 'fig']
g: ['gooseberry', 'grape', 'grapefruit', 'guava']
h: ['huckleberry', 'honeydew', 'honeydew']
j: ['jackfruit', 'jambul', 'jujube']
k: ['kiwi fruit', 'kumquat']
l: ['legume', 'lemon', 'lime', 'loquat', 'lychee']
m: ['mango', 'melon', 'mandarine']
w: ['watermelon',
 'williams pear or bartlett pear',
 'western raspberry (blackcap)',
 'watermelon-see melon']
r: ['rock melon', 'raspberry', 'rambutan', 'redcurrant']
n: ['nectarine', 'nut']
o: ['orange']
t: ['tangerine', 'tamarillo', 'tomato']
p: ['papaya',
 'passionfruit',
 'peach',
 'pepper',
 'pear',
 'persimmon',
 'physalis',
 'plum/prune (dried plum)',
 'pineapple',
 'pomegranate',
 'pomelo',
 'purple mangosteen']
q: ['quince']
s: ['salal berry', 'satsuma', 'star fruit', 'strawberry']
u: ['ugli fruit']
```

---

2 - Functional Programming

2a - Functions in depth

Question 1

Write a function called `read_news()` that accepts the following arguments:

- `time_of_day` (can be "morning", "noon, "afternoon", "evening" or "night". default: "morning")
- `source` (can be "online" or "paper". default: "online")

The function will validate that it got legal arguments and then print an output in the following format:

```
>>> read_news(time_of_day="noon", source="paper")
reading the news in the noon (source: paper)
```

Solution

In [122]:

```
def read_news(time_of_day="morning", source="online"):
    time_of_day_values = {"morning", "noon", "afternoon", "evening", "night"}
    source_values = {"online", "paper"}
    if time_of_day not in time_of_day_values:
        raise ValueError("unknown time")
    if source not in source_values:
        raise ValueError("unknown source")

    print("reading the news in the {} (source: {})".format(time_of_day, source))

read_news()
read_news(source="paper")
read_news(time_of_day="noon", source="paper")
read_news(source="online", time_of_day="evening")
```

```
reading the news in the morning (source: online)
reading the news in the morning (source: paper)
reading the news in the noon (source: paper)
reading the news in the evening (source: online)
```

Question 2

Write a function called `fruit_salad()` that accepts an unlimited number of fruits passed as string arguments.

The function will print it's output in the following format:

```
>>> fruit_salad("apple", "banana", "lychee", "orange")
fruit included:
===============
apple
banana
lychee
orange
```

Solution

In [124]:

```
def fruit_salad(*fruitargs):
    print("fruit included:\n" + "="*15)
#    for fruit in fruitargs:
#        print(fruit)
    print(*fruitargs, sep='\n')

fruit_salad("apple", "banana", "lychee", "orange")
```

```
fruit included:
===============
apple
banana
lychee
orange
```

Question 3

Document your solution to the Q1 with a docstring. Add a doctest for checking out a couple of call invocations.

Solution

In [125]:

```
def read_news(time_of_day="morning", source="online"):
    """A function for reading the news.

    This function deals with reading the news. It can read the news on
    different times of the day, and from different sources.

    Legal time_of_day values:
    "morning", "noon", "afternoon", "evening", "night"

    Legal source values:
    "online", "paper"

    Examples:
    >>> read_news()
    reading the news in the morning (source: online)

    >>> read_news(source="paper")
    reading the news in the morning (source: paper)

    >>> read_news(time_of_day="noon", source="paper")
    reading the news in the noon (source: paper)

    >>> read_news(source="online", time_of_day="evening")
    reading the news in the evening (source: online)

    >>> read_news(time_of_day="late")
    Error: unknown time of day

    >>> read_news(source="billboard")
    Error: unknown source
    """
    time_of_day_values = {"morning", "noon", "afternoon", "evening", "night"}
    source_values = {"online", "paper"}
    if time_of_day not in time_of_day_values:
        print("Error: unknown time of day")
        return
    if source not in source_values:
        print("Error: unknown source")
        return
    print("reading the news in the {} (source: {})".format(time_of_day, source))

import doctest
doctest.run_docstring_examples(read_newsws, globals(), verbose=True)
```

```
Finding tests in NoName
Trying:
    read_news()
Expecting:
    reading the news in the morning (source: online)
ok
Trying:
    read_news(source="paper")
Expecting:
    reading the news in the morning (source: paper)
ok
Trying:
    read_news(time_of_day="noon", source="paper")
Expecting:
    reading the news in the noon (source: paper)
ok
Trying:
    read_news(source="online", time_of_day="evening")
Expecting:
    reading the news in the evening (source: online)
ok
Trying:
    read_news(time_of_day="late")
Expecting:
    Error: unknown time of day
ok
Trying:
    read_news(source="billboard")
Expecting:
    Error: unknown source
ok
```

2b - Functional tools

Question 1

Use functional tools to write the code for the following requirements:

input: a list of strings

output: a list of tuples, each containing the chars from the matching string

Example

input: ['a', 'funny', 'story']

output: [('a',), ('f', 'u', 'n', 'n', 'y'), ('s', 't', 'o', 'r', 'y')]

Solution

In [66]:

```
data = ['a', 'funny', 'story']
res = list(map(tuple, data))
print(res)
```

```
[('a',), ('f', 'u', 'n', 'n', 'y'), ('s', 't', 'o', 'r', 'y')]
```

Question 2

Use the new functional tools for printing a list of strings, each one in a new line, without using a loop

Solution

In [129]:

```
data = ["a small story", "a short poem", "a long epilogue"]

list(map(print, data))

# OR:
#print(*data, sep='\n')
```

```
a small story
a short poem
a long epilogue
```

Out[129]:

```
[None, None, None]
```

Question 3

Read all the fruit name data from `files/2b-functional_tools/fruit.txt`, change them to upper case and print them. Do not use loops.

Solution

In [135]:

```
# Option 1 - using lambda function
with open('../files/2b-functional_tools/fruit.txt') as f:
    list(map(lambda x: print(x.strip().upper()), f))
```

APPLE
APRICOT
AVOCADO
BANANA
BREADFRUIT
BILBERRY
BLACKBERRY
BLACKCURRANT
BLUEBERRY
BOYSENBERRY
CANTALOUPE
CURRANT
CHERRY
CHERIMOYA
CHILI
CLOUDBERRY
COCONUT
DAMSON
DATE
DRAGONFRUIT
DURIAN
ELDERBERRY
FEIJOA
FIG
GOOSEBERRY
GRAPE
GRAPEFRUIT
GUAVA
HUCKLEBERRY
HONEYDEW
JACKFRUIT
JAMBUL
JUJUBE
KIWI FRUIT
KUMQUAT
LEGUME
LEMON
LIME
LOQUAT
LYCHEE
MANGO
MELON
CANARY MELON
CANTALOUPE
HONEYDEW
WATERMELON
ROCK MELON
NECTARINE
NUT
ORANGE
CLEMENTINE (TYPE OF ORANGE)
MANDARINE
TANGERINE
PAPAYA
PASSIONFRUIT
PEACH
PEPPER
PEAR
WILLIAMS PEAR OR BARTLETT PEAR
PERSIMMON

PLUM/PRUNE (DRIED PLUM)
PINEAPPLE
POMEGRANATE
POMELO
PURPLE MANGOSTEEN
QUINCE
RASPBERRY
WESTERN RASPBERRY (BLACKCAP)
RAMBUTAN
REDCURRANT
SALAL BERRY
SATSUMA
STAR FRUIT
STRAWBERRY
TAMARILLO
TOMATO
UGLI FRUIT
WATERMELON-SEE MELON

In [69]:

```
# Option 2 - using str functions
with open('../files/2b-functional_tools/fruit.txt') as f:
    list(map(print, map(str.upper, map(str.strip, f))))
```

APPLE
APRICOT
AVOCADO
BANANA
BREADFRUIT
BILBERRY
BLACKBERRY
BLACKCURRANT
BLUEBERRY
BOYSENBERRY
CANTALOUPE
CURRANT
CHERRY
CHERIMOYA
CHILI
CLOUDBERRY
COCONUT
DAMSON
DATE
DRAGONFRUIT
DURIAN
ELDERBERRY
FEIJOA
FIG
GOOSEBERRY
GRAPE
GRAPEFRUIT
GUAVA
HUCKLEBERRY
HONEYDEW
JACKFRUIT
JAMBUL
JUJUBE
KIWI FRUIT
KUMQUAT
LEGUME
LEMON
LIME
LOQUAT
LYCHEE
MANGO
MELON
CANARY MELON
CANTALOUPE
HONEYDEW
WATERMELON
ROCK MELON
NECTARINE
NUT
ORANGE
CLEMENTINE (TYPE OF ORANGE)
MANDARINE
TANGERINE
PAPAYA
PASSIONFRUIT
PEACH
PEPPER
PEAR
WILLIAMS PEAR OR BARTLETT PEAR
PERSIMMON

```
PLUM/PRUNE (DRIED PLUM)
PINEAPPLE
POMEGRANATE
POMELO
PURPLE MANGOSTEEN
QUINCE
RASPBERRY
WESTERN RASPBERRY (BLACKCAP)
RAMBUTAN
REDCURRANT
SALAL BERRY
SATSUMA
STAR FRUIT
STRAWBERRY
TAMARILLO
TOMATO
UGLI FRUIT
WATERMELON-SEE MELON
```

Question 4

Given the following list of tuples:

```
[('Apple', 16), ('Banana', 12), ('Orange', 7), ('Grapes', 30)]
```

Sort the list by the second item of the tuple.

Solution

In [70]:

```
import operator
from pprint import pprint

data = [('Apple', 16), ('Banana', 12), ('Orange', 7), ('Grapes', 30)]
sorted_list = sorted(data, key=operator.itemgetter(1))
pprint(sorted_list)
```

```
[('Orange', 7), ('Banana', 12), ('Apple', 16), ('Grapes', 30)]
```

Question 5

Given a list of strings (make your own for testing):

1

Use reduce to calculate the number of chars in all the strings

2

Use reduce and map to calculate the total number of chars

3

Did you use a function from the opertator module? if not do now :)

4

Modify your previous answer to use the builtin function `sum()` instead of reduce.

Solution

In [136]:

```
from functools import reduce

data = ["an invitation", "a book", "an orchid"]

# 1
res1 = reduce(lambda tot, curr: tot + len(curr), data, 0)

# 2
res2 = reduce(lambda a, b: a + b, map(len, data))

# 3
import operator
res3 = reduce(operator.add, map(len, data))

# 4
res4 = sum(map(len, data))

print("res 1 is", res1)
print("res 2 is", res2)
print("res 3 is", res3)
print("res 4 is", res4)
```

```
res 1 is 28
res 2 is 28
res 3 is 28
res 4 is 28
```

Question 6

Money bills have a number and a name of a famous important person. Define a MoneyBill class for holding that data.

Given the following money bill numbers: 20, 50, 10, 200

and matching names:

- Moshe Sharett
- Shaul Tchernichovsky
- Yizhak Ben-Zvi
- Zalman Shazar

Use `map()` to write functional code for creating all the MoneyBill instances (manually creating them doesn't count, your code should do it in a functional programming manner).

Solution

In [72]:

```
from collections import namedtuple
from pprint import pprint

MoneyBill = namedtuple("MoneyBill", "number, name")
numbers = [20, 50, 100, 200]
names = ["Moshe Sharett",
         "Shaul Tchernichovsky",
         "Yizhak Ben-Zvi",
         "Zalman Shazar"]

bills = map(MoneyBill, numbers, names)
pprint(list(bills))
```

```
[MoneyBill(number=20, name='Moshe Sharett'),
 MoneyBill(number=50, name='Shaul Tchernichovsky'),
 MoneyBill(number=100, name='Yizhak Ben-Zvi'),
 MoneyBill(number=200, name='Zalman Shazar')]
```

Question 7

Let's implement a function for calculating mathematical expressions of this sort, composed of numbers and add operator:

```
3 + 32 + 45 + 7
```

Our function will handle invalid expressions such as this:

```
3+++ 32
```

And this:

```
+ 25 +++6 +
```

1

Use imperative programming (with `if`s and `for` loops)

2

Solution

In [73]:

```
# 1
def calculate(exp):
    """
    >>> calculate("3 + 32 + 45 + 7")
    87
    >>> calculate("3+++ 32")
    35
    >>> calculate("+ 25 +++6 +")
    31
    """
    exp_parts = exp.split("+")
    res = 0
    for part in exp_parts:
        if part != "":
            res += int(part.strip())
    return res

import doctest
doctest.run_docstring_examples(calculate, globals())
```

In [74]:

```
# 2
from functools import reduce
import operator

def calculate(exp):
    """
    >>> calculate("3 + 32 + 45 + 7")
    87
    >>> calculate("3+++ 32")
    35
    >>> calculate("+ 25 +++6 +")
    31
    """
    return reduce(operator.add, map(int, filter(None, exp.split("+"))))

import doctest
doctest.run_docstring_examples(calculate, globals())
```

2c - Closures and decorators

Question 1

Implement a decorator called `timeme` for measuring the runtime of functions.

Use `time.time()` for measuring current time in seconds since the epoch.

Solution

In [ ]:

```
import time

def timeme(fn):
    def wrapper(*args, **kwargs):
        start = time.time()
        res = fn(*args, **kwargs)
        end = time.time()
        diff = (end - start) * 1000
        print("--- time passed: {:.3} milliseconds".format(diff))
    return wrapper
```

In [76]:

```
# testing

@timeme
def print_word(word="word"):
    print(word)

print_word('hello')
```

```
hello
--- time passed: 3.41e-05 milliseconds
```

In [77]:

```
@timeme
def calculate(n):
    return [i**3 for i in range(n)]

calculate(5000)
calculate(100000)
```

```
--- time passed: 0.00184 milliseconds
--- time passed: 0.0481 milliseconds
```

Question 2

For each of the HTML tags `div`, `p` and `strong`, implement a decorator for wrapping a function so that it will return it's regular output wrapped in the HTML tag.

1

Implement the decorators.

2

Try using more than one decorator on the target function, e.g.:

function:

```
def foo():
    return 'a yellow banana'
```

output:

```
<p><strong>a yellow banana</strong></p>
```

This is called **chaining decorators**.

Solution

In [78]:

```
def add_div(fn):
    def inner(*args, **kwargs):
        res = fn(*args, **kwargs)
        return "<div>{}</div>".format(res)
    return inner

def add_p(fn):
    def inner(*args, **kwargs):
        res = fn(*args, **kwargs)
        return "<p>{}</p>".format(res)
    return inner

def add_strong(fn):
    def inner(*args, **kwargs):
        res = fn(*args, **kwargs)
        return "<strong>{}</strong>".format(res)
    return inner
```

In [79]:

```
# test
@add_div
@add_p
@add_strong
def fruit_counter(name, number):
    return '{} - {} items'.format(name, number)

print(fruit_counter('avocado', 10))
print(fruit_counter('peach', 4))
```

```
<div><p><strong>avocado - 10 items</strong></p></div>
<div><p><strong>peach - 4 items</strong></p></div>
```

Question 3

Unite the decorators you wrote on the previous question to a single decorator that can support any type of HTML tag it gets as an argument.

Solution

In [80]:

```
def add_tag(fn, tag):
    def inner(*args, **kwargs):
        res = fn(*args, **kwargs)
        return "<{0}>{1}</{0}>".format(tag, res)
    return inner

def apple():
    return "apple"

decorated_apple = add_tag(add_tag(add_tag(add_tag(apple,
                                                  'a'),
                                        'p'),
                                  'div'),
                          'html')
print(decorated_apple())
```

```
<html><div><p><a>apple</a></p></div></html>
```

In [81]:

```
# Using the configurable decorator syntax
def add_tag(tag):
    def wrapper(fn):
        def inner(*args, **kwargs):
            res = fn(*args, **kwargs)
            return "<{0}>{1}</{0}>".format(tag, res)
        return inner
    return wrapper
```

In [82]:

```
# test
@add_tag("body")
@add_tag("div")
@add_tag("p")
@add_tag("a")
def apple():
    return "apple"

print(apple())
```

```
<body><div><p><a>apple</a></p></div></body>
```

Question

Write a decorator called `authorisation` that receives a string parameter. Implement the decorator so that only if the correct authorisation is supplied to the decorated function.

3 - Objects and Classes

3a - Creating classes, inheritance

Question 1

Let's implement an anti-theft car coded lock system ("CODAN").

1

Implement an `unlock()` function that will ask the user for the car lock and check if it matches the real code. Use `input()` to get the code from the user. Allow up to 3 guesses before refusing.

Solution

In [83]:

```
CORRECT_CODE = "4434"
NUM_OF_TRIES = 3

def unlock():
    for i in range(NUM_OF_TRIES):
        code = input('Please enter code: ')
        if code == CORRECT_CODE:
            return True
    return False

unlock()
```

```
Please enter code: 4434
```

Out[83]:

```
True
```

2

Our previous implementation is good for checking the lock code, but now we want to increase the security of the system, and following 3 successive wrong guesses, lock the system down. Modify your solution to the previous question so that you will have a `CarLock` class that will keep the lock state of the system (*Locked* (default), *Unlocked* (after successful `unlock()`) and *Blocked* after too many tries). When Blocked, the system should not allow further unlock tries.

Solution

In [84]:

```
NUM_OF_TRIES = 3
CORRECT_CODE = "4434"

class LockState(object):
    LOCKED, UNLOCKED, BLOCKED = range(3)


class CarLock(object):
    def __init__(self):
        self._lock_state = LockState.LOCKED

    def unlock(self):
        if self._lock_state == LockState.BLOCKED:
            return False
        if self._lock_state == LockState.UNLOCKED:
            return True
        for _ in range(NUM_OF_TRIES):
            code = input('Please enter code: ')
            if code == CORRECT_CODE:
                self._lock_state = LockState.UNLOCKED
                return True
        self._lock_state = LockState.BLOCKED
        return False

    def is_locked(self):
        return self._lock_state != LockState.UNLOCKED

c = CarLock()
c.unlock()
print("locked?", c.is_locked())
c.unlock()
print("locked?", c.is_locked())
```

```
Please enter code: 4434
locked? False
locked? False
```

3

Modify your solution to make it more general - allow trying for different number of tries (default is 3). Also allow using different "correct passwords" for different lock instances.

How will you implement these changes in your solution?

Solution

In [85]:

```
class LockState(object):
    LOCKED, UNLOCKED, BLOCKED = range(3)


class CarLock(object):
    def __init__(self, correct_code, num_of_tries=3):
        self._num_of_tries = num_of_tries
        self._correct_code = correct_code
        self._lock_state = LockState.LOCKED

    def unlock(self):
        if self._lock_state == LockState.BLOCKED:
            return False
        if self._lock_state == LockState.UNLOCKED:
            return True
        for _ in range(self._num_of_tries):
            code = input('Please enter code: ')
            if code == self._correct_code:
                self._lock_state = LockState.UNLOCKED
                return True
        self._lock_state = LockState.BLOCKED
        return False

    def is_locked(self):
        return self._lock_state != LockState.UNLOCKED

c = CarLock(num_of_tries=4, correct_code="4434")
c.unlock()
print("locked?", c.is_locked())
c.unlock()
print("locked?", c.is_locked())
```

```
Please enter code: 4434
locked? False
locked? False
```

4

Factor out the Input device used to insert the code into the car lock system to add support for numeric keypads and alphanumeric (letters and digits) keyboard ones that can have letters and other characters too.

Implement a `NumericKeypadInput` class and an `AlphaNumericInput` class. Both classes will implement the `get_input()` method and run validation to make sure the input they received matches their limitations. If an illegal value is detected raise a `ValueError` exception.

Make the necessary changes so that your `CarLock` class will support the new input devices.

Solution

In [86]:

```
class InputDevice(object):
    def get_input(self):
        data = input('Please enter code: ')
        if not self._validate_input(data):
            raise ValueError("Error: invalid input: {}".format(data))
        return data

    def _validate_input(self, data):
        return True


class NumericKeypadInput(InputDevice):
    def _validate_input(self, data):
        return data.isdigit()


class AlphaNumericInput(InputDevice):
    def _validate_input(self, data):
        return data.isalnum()


class LockState(object):
    LOCKED, UNLOCKED, BLOCKED = range(3)


class CarLock(object):
    def __init__(self, correct_code, input_device, num_of_tries=3):
        self._num_of_tries = num_of_tries
        self._correct_code = correct_code
        self._input_device = input_device
        self._lock_state = LockState.LOCKED

    def unlock(self):
        if self._lock_state == LockState.BLOCKED:
            return False
        if self._lock_state == LockState.UNLOCKED:
            return True
        for _ in range(self._num_of_tries):
            try:
                code = self._input_device.get_input()
                if code == self._correct_code:
                    self._lock_state = LockState.UNLOCKED
                    return True
            except ValueError:
                print("Error: Illegal input detected")
        self._lock_state = LockState.BLOCKED
        return False

    def is_locked(self):
        return self._lock_state != LockState.UNLOCKED

c = CarLock(num_of_tries=4, input_device=NumericKeypadInput(),
            correct_code="4434")
c.unlock()
print("locked?", c.is_locked())
c.unlock()
print("locked?", c.is_locked())
```

```
Please enter code: 1151
locked? False
locked? False
```

5

Implement a `CodeProtectionDevice` class that will handle the aspects of validating a provided pass code against the actual correct code. Make your `CarLock` system inherit from both `CodeProtectionDevice` and one of the input devices.

Implement `NumericCarLock` and `AlphanumericCarLock` classes that will extend the new `CodeProtectionDevice`.

Solution

In [87]:

```
class InputDevice(object):
    def get_input(self):
        data = input('Please enter code: ')
        if not self._validate_input(data):
            raise ValueError("Error: invalid input: {}".format(data))
        return data

    def _validate_input(self, data):
        return True


class NumericKeypadInput(InputDevice):
    def _validate_input(self, data):
        return data.isdigit()


class AlphaNumericInput(InputDevice):
    def _validate_input(self, data):
        return data.isalnum()


class LockState(object):
    LOCKED, UNLOCKED, BLOCKED = range(3)


class CodeProtectionDevice(object):
    def __init__(self, correct_code, input_device, num_of_tries=3):
        self._num_of_tries = num_of_tries
        self._correct_code = correct_code
        self._input_device = input_device
        self._lock_state = LockState.LOCKED

    def unlock(self):
        if self._lock_state == LockState.BLOCKED:
            return False
        if self._lock_state == LockState.UNLOCKED:
            return True
        for i in range(self._num_of_tries):
            try:
                code = self._input_device.get_input()
                if code == self._correct_code:
                    self._lock_state = LockState.UNLOCKED
                    return True
            except ValueError:
                print("Error: Illegal input detected")
        self._lock_state = LockState.BLOCKED
        return False

    def is_locked(self):
        return self._lock_state != LockState.UNLOCKED

class NumericCarLock(CodeProtectionDevice):
    def __init__(self, correct_code, num_of_tries=3):
        super(NumericCarLock, self).__init__(correct_code,
                                             num_of_tries=num_of_tries,
                                             input_device=NumericKeypadInput())

class AlphaNumericCarLock(CodeProtectionDevice):
```

```
        super(NumericCarLock, self).__init__(correct_code,
                                             num_of_tries=num_of_tries,
                                             input_device=AlphaNumericInput())

c = NumericCarLock(num_of_tries=4, correct_code="4434")
c.unlock()
print("locked?", c.is_locked())
c.unlock()
print("locked?", c.is_locked())
```

```
Please enter code: 4434
locked? False
locked? False
```

3b - Operator overloading, properties, abstract base classes

Question 1

Revisiting our car lock system, expose the following data as properties:

- Number of permitted tries
- Is the system locked

Solution

In [88]:

```
class InputDevice(object):
    def get_input(self):
        data = input('Please enter code: ')
        if not self._validate_input(data):
            raise ValueError("Error: invalid input: {}".format(data))
        return data

    def _validate_input(self, data):
        return True


class NumericKeypadInput(InputDevice):
    def _validate_input(self, data):
        return data.isdigit()


class AlphaNumericInput(InputDevice):
    def _validate_input(self, data):
        return data.isalnum()


class LockState(object):
    LOCKED, UNLOCKED, BLOCKED = range(3)


class CodeProtectionDevice(object):
    def __init__(self, correct_code, input_device, num_of_tries=3):
        self._num_of_tries = num_of_tries
        self._correct_code = correct_code
        self._input_device = input_device
        self._lock_state = LockState.LOCKED

    def unlock(self):
        if self._lock_state == LockState.BLOCKED:
            return False
        if self._lock_state == LockState.UNLOCKED:
            return True
        for i in range(self._num_of_tries):
            try:
                code = self._input_device.get_input()
                if code == self._correct_code:
                    self._lock_state = LockState.UNLOCKED
                    return True
            except ValueError:
                print("Error: Illegal input detected")
        self._lock_state = LockState.BLOCKED
        return False

    @property
    def permitted_tries(self):
        return self._num_of_tries

    @property
    def is_locked(self):
        return self._lock_state != LockState.UNLOCKED

class NumericCarLock(CodeProtectionDevice):
    def __init__(self, correct_code, num_of_tries=3):
```

```
                                                num_of_tries=num_of_tries,
                                                input_device=NumericKeypadInput())

class AlphaNumericCarLock(CodeProtectionDevice):
    def __init__(self, correct_code, num_of_tries=3):
        super(NumericCarLock, self).__init__(correct_code,
                                                num_of_tries=num_of_tries,
                                                input_device=AlphaNumericInput())

c = NumericCarLock(num_of_tries=4, correct_code="4434")
c.unlock()
print("permitted tries are {}, locked? {}".format(c.permitted_tries,
                                                  c.is_locked))
c.unlock()
print("permitted tries are {}, locked? {}".format(c.permitted_tries,
                                                  c.is_locked))
```

```
Please enter code: 4434
permitted tries are 4, locked? False
permitted tries are 4, locked? False
```

Question 2

Notice that both input devices defined earlier share the same basic behaviour. Factor out a common abstract base class, and make the two input devices implement it.

Solution

In [89]:

```
import abc

class InputDevice(abc.ABC):
    def get_input(self):
        data = input('Please enter code: ')
        if not self._validate_input(data):
            raise ValueError("Error: invalid input: {}".format(data))
        return data

    @abc.abstractmethod
    def _validate_input(self, data):
        return True


class NumericKeypadInput(InputDevice):
    def _validate_input(self, data):
        return data.isdigit()


class AlphaNumericInput(InputDevice):
    def _validate_input(self, data):
        return data.isalnum()


class LockState(object):
    LOCKED, UNLOCKED, BLOCKED = range(3)


class CodeProtectionDevice(object):
    def __init__(self, correct_code, input_device, num_of_tries=3):
        self._num_of_tries = num_of_tries
        self._correct_code = correct_code
        self._input_device = input_device
        self._lock_state = LockState.LOCKED

    def unlock(self):
        if self._lock_state == LockState.BLOCKED:
            return False
        if self._lock_state == LockState.UNLOCKED:
            return True
        for i in range(self._num_of_tries):
            try:
                code = self._input_device.get_input()
                if code == self._correct_code:
                    self._lock_state = LockState.UNLOCKED
                    return True
            except ValueError:
                print("Error: Illegal input detected")
        self._lock_state = LockState.BLOCKED
        return False

    @property
    def permitted_tries(self):
        return self._num_of_tries

    @property
    def is_locked(self):
        return self._lock_state != LockState.UNLOCKED
```

```
class NumericCarLock(CodeProtectionDevice):
    def __init__(self, correct_code, num_of_tries=3):
        super(NumericCarLock, self).__init__(correct_code,
                                             num_of_tries=num_of_tries,
                                             input_device=NumericKeypadInput())

class AlphaNumericCarLock(CodeProtectionDevice):
    def __init__(self, correct_code, num_of_tries=3):
        super(NumericCarLock, self).__init__(correct_code,
                                             num_of_tries=num_of_tries,
                                             input_device=AlphaNumericInput())

c = NumericCarLock(num_of_tries=4, correct_code="4434")
c.unlock()
print("permitted tries are {}, locked? {}".format(c.permitted_tries, c.is_locked
))
c.unlock()
print("permitted tries are {}, locked? {}".format(c.permitted_tries, c.is_locked
))
```

```
Please enter code: 4434
permitted tries are 4, locked? False
permitted tries are 4, locked? False
```

Question 3 (Bonus)

The inputDevice classes can be seen as deliveing capabilities to the `CodeProtectionDevice` classes. Lookup the design pattern *Mixin* and implement the input devices in this manner as part of `NumericCarLock` and `AlphaNumericCarLock`.

---

4 - Iterators

4a - List comprehension

Question 1

Use `input()` to ask the user for a fruit name prefix. Iterate over the fruit names from `files/3a-list_comprehension/fruit.txt` and print in upper case the ones that start with the select prefix.

Solution

In [90]:

```
with open('../files/2b-functional_tools/fruit.txt') as f:
    prefix = input('Please enter the fruit name prefix: ')
    res = []
    for fruit in f:
        if fruit.lower().startswith(prefix):
            print(fruit.strip().upper())
```

```
Please enter the fruit name prefix: ap
APPLE
APRICOT
```

Question 2

Perform the previous task, now using `map()` and `filter()` to prepare a list containing the matching fruits in upper case.

Solution

In [91]:

```
with open('../files/2b-functional_tools/fruit.txt') as f:
    prefix = input('Please enter the fruit name prefix: ')
    matching_fruit = filter(lambda x: x.lower().startswith(prefix), f)
    upper_case_fruit = map(lambda x: x.strip().upper(), matching_fruit)
    print(list(upper_case_fruit))
```

```
Please enter the fruit name prefix: ap
['APPLE', 'APRICOT']
```

Question 3

Now solve the previous question again using list comprehension.

Solution

In [92]:

```
with open('../files/2b-functional_tools/fruit.txt') as f:
    prefix = input('Please enter the fruit name prefix: ')
    res = [
        fruit.strip().upper()
        for fruit in f
        if fruit.lower().startswith(prefix)
    ]
    print(res)
```

```
Please enter the fruit name prefix: ap
['APPLE', 'APRICOT']
```

Question 4

Given a text implement a function that will return a list of all the words in the text that do not contain only digits.

Solution

In [93]:

```
def not_only_digits(data):
    return [
        word
        for word in data.split()
        if not word.isdigit()
    ]
```

In [94]:

```
not_only_digits('a funny story I heard involves 12 monkeys with 3eyes')
```

Out[94]:

```
['a', 'funny', 'story', 'I', 'heard', 'involves', 'monkeys', 'with',
'3eyes']
```

4b - Generator expressions

Question 1

1

Implement a generator expression for iterating over the numbers 0-5000 and calculating for each number its value to the power of 3 (2 --> 8, 3 --> 27, etc.)

Solution

In [95]:

```
cubes = (i**3 for i in range(5000))

for i in range(10):
    print(next(cubes))
```

```
0
1
8
27
64
125
216
343
512
729
```

4c - Generator functions

Question

Write a function that will receive a string as a parameter and print some data about the string.

1

For the following input:

```
a man a plan a canal panama
```

Implement a function that will print the following:

```
Word number 0 is 1 characters long
Word number 1 is 3 characters long
Word number 2 is 1 characters long
Word number 3 is 4 characters long
Word number 4 is 1 characters long
Word number 5 is 5 characters long
Word number 6 is 6 characters long
```

Solution

In [96]:

```
def print_word_data(data):
    """
    >>> print_word_data('a man a plan a canal panama')
    Word number 0 is 1 characters long
    Word number 1 is 3 characters long
    Word number 2 is 1 characters long
    Word number 3 is 4 characters long
    Word number 4 is 1 characters long
    Word number 5 is 5 characters long
    Word number 6 is 6 characters long
    """
    words = data.split()
    for i in range(len(words)):
        word = words[i]
        print("Word number {} is {} characters long".format(i, len(word)))

import doctest
doctest.run_docstring_examples(print_word_data, globals())
```

In [97]:

```
print_word_data('a man a plan a canal panama')
```

```
Word number 0 is 1 characters long
Word number 1 is 3 characters long
Word number 2 is 1 characters long
Word number 3 is 4 characters long
Word number 4 is 1 characters long
Word number 5 is 5 characters long
Word number 6 is 6 characters long
```

2

The `enumerate()` function is very useful for iterating over sequences, when you need both the item and the current index. Try running the following for loop to see how it behaves:

```
for i, item in enumerate(['first', 'second', 'third', 'last']):
    print(i, item)
```

Change your solution so that it will now use `enumerate()`.

Solution

In [98]:

```
def print_word_data(data):
    """
    >>> print_word_data('a man a plan a canal panama')
    Word number 0 is 1 characters long
    Word number 1 is 3 characters long
    Word number 2 is 1 characters long
    Word number 3 is 4 characters long
    Word number 4 is 1 characters long
    Word number 5 is 5 characters long
    Word number 6 is 6 characters long
    """
    words = data.split()
    for i, word in enumerate(words):
        print("Word number {} is {} characters long".format(i, len(word)))


import doctest
doctest.run_docstring_examples(print_word_data, globals())
```

3

Implement `my_enumerate` that will duplicate the behaviour of `enumerate()`. Design your solution as a class that implements the iterator protocol we discussed in class.

Solution

In [99]:

```
class my_enumerate(object):
    """
    >>> list(my_enumerate(['a', 'b', 'c']))
    [(0, 'a'), (1, 'b'), (2, 'c')]
    >>> list(my_enumerate([]))
    []
    """
    def __init__(self, data):
        self._data = data
    def __iter__(self):
        self._curr = 0
        self._iter = iter(self._data)
        return self
    def __next__(self):
        res = self._curr, next(self._iter)
        self._curr += 1
        return res

import doctest
doctest.run_docstring_examples(my_enumerate, globals())
```

4

Write yet another enumerate construct, this time called `my_gen_enumerate` and implement the enumeration using a generator function.

Solution

In [100]:

```
def my_gen_enumerate(data):
    """
    >>> list(my_enumerate(['a', 'b', 'c']))
    [(0, 'a'), (1, 'b'), (2, 'c')]
    >>> list(my_enumerate([]))
    []
    """
    curr = 0
    for item in data:
        yield curr, item
        curr += 1

import doctest
doctest.run_docstring_examples(my_gen_enumerate, globals())
```

4d - Iteration tools

Question 1

Write code that will return a generator object that will return all the even numbers starting with 0.

Solution

In [101]:

```
import itertools
def even_count():
    return (a for a in itertools.count() if a %2 == 0)
```

In [102]:

```
e = even_count()
for i in range(10):
    print(next(e))
```

```
0
2
4
6
8
10
12
14
16
18
```

Question 2

We saw in class how a for loop uses iterators for running.

Implement a for loop function that receives:

- a sequence to iterate over
- a function to be called on each item in the sequence

Solution

In [103]:

```
def for_func(sequence, func):
    it = iter(sequence)
    while True:
        try:
            item = next(it)
            func(item)
        except StopIteration:
            break
```

In [104]:

```
for_func([1, 2, 3], lambda x: print(x+200))
```

```
201
202
203
```

Question (hard)

Let's revisit our code from question 1 in 2b - Functional tools. In that question you implemented code that returns a list of tuples, each containing the chars of a word.

Let's write code that will take that list and return an iterator for it that returns each tuple as a concatenated word. For example:

Input: [('a',), ('f', 'u', 'n', 'n', 'y'), ('s', 't', 'o', 'r', 'y')]

Output: an iterator `sm` so that:

```
>>> next(sm)
'a'
>>> next(sm)
'funny'
>>> next(sm)
'story'
>>> next(sm)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Solution

In [105]:

```
import itertools
import operator

data = 'a funny story'.split()
data1 = map(tuple, data)
sm = itertools.starmap(lambda *x: reduce(operator.add, x), data1)
try:
    while True:
        print(next(sm))
except StopIteration:
    print('--- Reached the end ---')
```

```
a
funny
story
--- Reached the end ---
```

---

5 - Advanced Features

Question 1

When writing scripts that work on files, you have to change directories quite a lot. Different parts of the script may need to run from different directories, so you could end up entering one, doing something, returning back, entering another directory, and so on.

Write a context manager `cd` that will enable you to run specific commands inside a given directory, and then return automatically to the origin directory after finishing those commands.

The usage will be like this:

```
run_external_command(...)      # in origin directory
with cd('/tmp/fruit'):
    run_external_command(...)  # inside /tmp/fruit
run_external_command(...)      # in origin directory
```

Familiarize yourself with the following functions from the `os` module:

- `chdir()` changes directories
- `getcwd()` returns the current working directory

Solution

In [106]:

```
from contextlib import contextmanager
import os

@contextmanager
def cd(new_dir):
    origin_dir = os.getcwd()
    try:
        os.chdir(new_dir)
        yield
    finally:
        os.chdir(origin_dir)
```

In [107]:

```
print('before: {}'.format(os.getcwd()))
with cd('/tmp'):
    print('during: {}'.format(os.getcwd()))
print('after: {}'.format(os.getcwd()))
```

```
before: /Users/amit/Documents/Advanced Python Course/Labs/exercise a
nd solutions
during: /private/tmp
after: /Users/amit/Documents/Advanced Python Course/Labs/exercise an
d solutions
```

Question 2

Implement a context manager `timeme` for measuring the time a task takes.

The contextmanager will take a parameter `task_name` and will print the amount of time that passed while performing the task.

Example usage:

```
with timeme("calculation"):
    calculate_something()
```

Output:

```
--- the calculation task took 1.037 milliseconds
```

Solution

In [108]:

```
from contextlib import contextmanager
import time

@contextmanager
def timeme(task_name='default'):
    start = time.time()
    try:
        yield
    finally:
        stop = time.time()
        time_in_millis = (stop - start) * 1000
        print('--- the {} task took {:0.4} milliseconds'.format(task_name, time_
in_millis))
```

In [109]:

```
with timeme("sum of list comprehension"):
    print(sum([i**3 for i in range(5000)]))
```

```
156187506250000
--- the sum of list comprehension task took 1.838 milliseconds
```

---

6 - Testing

Question 1

Implement a class `Calculator` with a function called `average()` that receives a list of numbers and return their average. Write unit tests for this function, with a couple of test functions - each one testing a different condition.

Solution

In [110]:

```
class Calculator(object):
    def average(self, data):
        if len(data) == 0:
            return 0
        return sum(data) / len(data)
```

In [111]:

```
import unittest

class CalculatorTest(unittest.TestCase):
    def setUp(self):
        self.c = Calculator()

    def test_calculation(self):
        data = [11, 22, 33, 44, 55]
        self.assertEqual(self.c.average(data), 33)

    def test_empty(self):
        data = []
        self.assertEqual(self.c.average(data), 0)

if __name__ == "__main__":
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
..
----------------------------------------------------------------------
--
Ran 2 tests in 0.002s

OK
```

INTERNAL - Running all doctests

In [112]:

```
import doctest
# doctest.testmod(verbose=True)
doctest.testmod()
```

Out[112]:

```
TestResults(failed=0, attempted=11)
```