Advanced Python Labs

---

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.

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.

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

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 it's constructor method. Look back at tuple unpacking for more information.

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

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)
```

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
```

Question 3

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

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')]

Question 2

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

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.

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.

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.

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).

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 ifs and for loops)

2

Use functional programming (no ifs or loops)

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.

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**.

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.

---

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.

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.

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?

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 `getInput()` 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.

5

Implement a `CodeProtectionDevice` class that will handle the aspects of validating a provided pass code agains 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`.

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

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.

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. Read all the fruit name data from `files/3a-list_comprehension/fruit.txt` and print in upper case the ones that start with the select prefix.

Question 2

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

Question 3

Now solve the previous question again without using list comprehension.

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.

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.)

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
```

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

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

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.

4

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

4d - Iteration tools

Question 1

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

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

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
```

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

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
```

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.