Exercise Sheet 05 Solutions – Handling Errors and Debugging

Sebastian Höffner Aline Vilks

Deadline: Mon, 8 May 2017 08:00 +0200

Exercise 1: Reversed class room

Here are some annotations about what is wrong.

File: `whatdoesitdo_annotated.py`

```
"""
This module is a simple database handler.
"""
DATABASE_FILE = 'recorded_persons'
NAME = 'name'
AGE = 'age'
HEIGHT = 'height'
VALUE_SEPARATOR = ','


def read_database():
    """Reads the database file.

    Reads the database file line by line. Each line is split at the
    value separator into three parts:
        name   a string
        age    an integer
        height a float
    A person-dictionary is created from the three values.

    Returns:
        A list of all persons.
    """
    # @shoeffner: missing creation of persons = []
    with open(DATABASE_FILE, 'r') as file:
        for line in file.read().splitlines():
```

```
            values = line.split(VALUE_SEPARATOR)
            # @shoeffner: nam is wrong, better: NAME
            # @shoeffner: better: AGE
            # @shoeffner: better: HEIGHT
            person = {'nam': values[0],
                      'age': int(values[1]),
                      'height': float(values[2])}
            persons.append(person)
    return persons


def read_records(conditions):
    persons = read_database()
    # @shoeffner: Missing :
    if '*' in conditions:
        # @shoeffner: break is wrong, must be return
        break persons
    for condition in conditions:
        # @shoeffner: no whitespace in the next rows
        operation=condition.split(' ')
        key=operation[0]
        operator=operation[1]
        test_value=operation[2]
        persons=filter_records(persons,key,operator,test_value)
    return persons
# @shoeffner: not enough empty lines
def filter_records(records, key, operator, test_value):
    """Removes all records from a list of records, which do not
    fulfill the specified criteria.
    """
    filtered_records = []
    for record in records:
        # @shoeffner: missing , and :
        if isinstance(record[key] int)
            test_value = int(test_value)
        if isinstance(record[key], float):
            test_value = float(test_value)
        # @shoeffner: swapped operators
        if (operator == '<' and record[key] > test_value) \
                # @shoeffner: swapped operators, missing \
                or (operator == '>' and record[key] < test_value)
                or (operator == '==' and record[key] == test_value) \
                # @shoeffner: == is wrong here - copy/paste?
                or (operator == '!=' and record[key] == test_value):
            filtered_records.append(record)
```

```
    return filtered_records
# @shoeffner: not enough empty lines
def write_record(record):
    # @shoeffner: no whitespace + file mode must be 'a' not 'w'
    with open(DATABASE_FILE,'w')as file:
        print(record[NAME], record[AGE], record[HEIGHT],
              # @shoeffner: separator has to be , not ;. Use VALUE_SEPARATOR
              sep=';', file=file)
    return record


def print_records(records):
    print()
    while record in records:
        print(NAME, record[NAME], end='  ', sep=': ')
        print(AGE, record[AGE], end='  ', sep=': ')
        print(HEIGHT, record[HEIGHT], sep=': ')
    print()


def ask(question):
    return input(question + ' ')


def get_conditions():
    condition_string = ask('Which persons do you want to select? (Condition)')
    conditions = condition_string.split(' and ')
    # @shoeffner: return value is missing! return conditions
    return


def create_record():
    # @shoeffner: will never enter loop, use e.g. 'no'
    correct = 'yes'
    while correct[0].lower() != 'y':
        name = ask('Name?')
        age = ask('Age?')
        age = ask('Height?')
        record = {NAME: name, AGE: age, HEIGHT: height}
        # @shoeffner: print_records: wrong spelling + takes list
        pritn_records(record)
        correct = ask('Correct? [yes/no]')
    return record
```

```
def main():
    """Runs the program.

    Until the users enters 'q' to exit, they can use the following commands
    to interact with the database:
        a: add a new entry
        r: read entries
        q: exit
        d (hidden): add example data
    """
    response = ''
    while response != 'q':
        response = ask('Do you want to add records, retrieve records, or ' +
                       'quit? [a, r, q]')
        if response == 'r':
            conditions = get_conditions()
            records = read_records(conditions)
            print_records(records)
        elif response == 'a':
            record = create_record()
            write_record(record)
    # @shoeffner: indentation level wrong
    elif response == 'd':
        fill_with_example_data()


def fill_with_example_data():
    """Writes example data into the database."""
    records = [
        {NAME: 'Graham', AGE: 48, HEIGHT: 1.88},
        {NAME: 'John', AGE: 77, HEIGHT: 1.96},
        {NAME: 'Terry', AGE: 76, HEIGHT: 1.75},
        {NAME: 'Eric', AGE: 74, HEIGHT: 1.85},
        {NAME: 'Terry', AGE: 75, HEIGHT: 1.73},
        {NAME: 'Michael', AGE: 73, HEIGHT: 1.78}
    ]
    for record in records:
        write_record(record)


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

And some possible corrections and documentation comments.

File: `people_database.py`

```
"""
This module is a simple database handler.
It allows to manage persons as dictionaries in a database file.

Each person is represented as a dictionary containing three values:
    a name (string), an age (integer), a height (float)

To read from the database, a very simple query language is implemented:

    query := '*' | key <space> operator <space> test
           | query <space> 'and' <space> query
    key := 'name' | 'age' | 'height'
    operator := '<' | '>' | '==' | '!='
    test := <string, float, or int value>

When the query contains '*', all records are returned. Otherwise, the records
are filtered by the specified query criteria, such that all must be fulfilled
for a record to be included in the result list.

The database represents records as comma seperated values and linebreak
separated rows. No escaping is done.
"""
DATABASE_FILE = 'recorded_persons'
NAME = 'name'
AGE = 'age'
HEIGHT = 'height'
VALUE_SEPARATOR = ','


def read_database():
    """Reads the database file.

    Reads the database file line by line. Each line is split at the
    value separator into three parts:
        name   a string
        age    an integer
        height a float
    A person-dictionary is created from the three values.

    Returns:
        A list of all persons.
    """
    persons = []
    with open(DATABASE_FILE, 'r') as file:
        for line in file.read().splitlines():
```

```
            values = line.split(VALUE_SEPARATOR)
            person = {NAME: values[0],
                      AGE: int(values[1]),
                      HEIGHT: float(values[2])}
            persons.append(person)
    return persons


def read_records(conditions):
    """Reads the records from the database which fulfill all conditions.

    If conditions includes '*', all records are returned.
    Otherwise only those records matching all conditions are returned.

    Args:
        conditions: A list of conditions to be fulfilled by a record.

    Returns:
        The list of matching records.
    """
    persons = read_database()
    if '*' in conditions:
        return persons
    for condition in conditions:
        operation = condition.split(' ')
        key = operation[0]
        operator = operation[1]
        test_value = operation[2]
        persons = filter_records(persons, key, operator, test_value)
    return persons


def filter_records(records, key, operator, test_value):
    """Removes all records from a list of records, which do not
    fulfill the specified criteria.

    The criteria are checked by applying the operator with the record's value
    for the specified key as the first and the test_value as the second
    operand.

    If the record's value is not a string, int and float casting are tried
    for the test_value.

    Args:
        records: The list of records.
```

```
        key: The key to select from the record.
        operator: The relation to test.
        test_value: The value to test against.

    Returns:
        A list of records for which all criteria are True.
    """
    filtered_records = []
    for record in records:
        if isinstance(record[key], int):
            test_value = int(test_value)
        if isinstance(record[key], float):
            test_value = float(test_value)
        if (operator == '<' and record[key] < test_value) \
                or (operator == '>' and record[key] > test_value) \
                or (operator == '==' and record[key] == test_value) \
                or (operator == '!=' and record[key] != test_value):
            filtered_records.append(record)
    return filtered_records


def write_record(record):
    """Writes a record to the database.

    Args:
        record: The record to write.

    Returns:
        The record that was written.
    """
    with open(DATABASE_FILE, 'a') as file:
        print(record[NAME], record[AGE], record[HEIGHT],
              sep=VALUE_SEPARATOR, file=file)
    return record


def print_records(records):
    """Prints a list of records.

    Args:
        records: The list of records.
    """
    print()
    for record in records:
        print(NAME, record[NAME], end='  ', sep=': ')
```

```
        print(AGE, record[AGE], end='  ', sep=': ')
        print(HEIGHT, record[HEIGHT], sep=': ')
    print()


def ask(question):
    """Asks a question.

    Appends a blank space after the question.

    Args:
        question: The question to prompt.

    Returns:
        The user input.
    """
    return input(question + ' ')


def get_conditions():
    """Asks for conditions and splits them into a list of conditions.

    The conditions are split around ' and '.

    Returns:
        A list containing conditions.
    """
    condition_string = ask('Which persons do you want to select? (Condition)')
    conditions = condition_string.split(' and ')
    return conditions


def create_record():
    """Creates a record from user input.

    Asks the user for name, age, and height. Then prints the possible record.
    Only if the user accepts with 'y' or 'Y', the record is returned.
    Otherwise, the user is asked again.

    Returns:
        A dictionary representing a person.
    """
    correct = 'no'
    while correct[0].lower() != 'y':
        name = ask('Name?')
```

```
        age = ask('Age?')
        height = ask('Height?')
        record = {NAME: name, AGE: age, HEIGHT: height}
        print_records([record])
        correct = ask('Correct? [yes/no]')
    return record


def main():
    """Runs the program.

    Until the users enters 'q' to exit, they can use the following commands
    to interact with the database:
        a: add a new entry
        r: read entries
        q: exit
        d (hidden): add example data
    """
    response = ''
    while response != 'q':
        response = ask('Do you want to add records, retrieve records, or ' +
                       'quit? [a, r, q]')
        if response == 'r':
            conditions = get_conditions()
            records = read_records(conditions)
            print_records(records)
        elif response == 'a':
            record = create_record()
            write_record(record)
        elif response == 'd':
            fill_with_example_data()


def fill_with_example_data():
    """Writes example data into the database."""
    records = [
        {NAME: 'Graham', AGE: 48, HEIGHT: 1.88},
        {NAME: 'John', AGE: 77, HEIGHT: 1.96},
        {NAME: 'Terry', AGE: 76, HEIGHT: 1.75},
        {NAME: 'Eric', AGE: 74, HEIGHT: 1.85},
        {NAME: 'Terry', AGE: 75, HEIGHT: 1.73},
        {NAME: 'Michael', AGE: 73, HEIGHT: 1.78}
    ]
    for record in records:
        write_record(record)
```

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

Exercise 2: Reading and writing csv files

File: `iris_correction.py`

```
with open('iris.data', 'r') as iris_in, open('iris.csv', 'w') as iris_out:
    for i, line in enumerate(iris_in, 1):
        if i == 35:
            line = '4.9,3.1,1.5,0.2,Iris-setosa\n'
        if i == 38:
            line = '4.9,3.6,1.4,0.1,Iris-setosa\n'
        print(line, file=iris_out, end='')
```

For the statistics you could either build your own solution (and add `asserts`) or use the `statistics` module. We added both solutions into our example.

File: `iris_statistics.py`

```
"""
Prints some statistics about the iris data set.
"""
import csv
import os
import statistics


IRIS_FILE = 'iris.csv'
SEPAL_LENGTH = 'sepal length in cm'
SEPAL_WIDTH = 'sepal width in cm'
PETAL_LENGTH = 'petal length in cm'
PETAL_WIDTH = 'petal width in cm'
CLASS = 'class'


def read_with_csv(filename):
    """Reads the iris file using the csv module.

    Args:
        filename: The filename.

    Returns:
        A list of dictionaries mapping the features to their values.
```

```
    """
    with open(filename, 'r') as iris_file:
        reader = csv.DictReader(iris_file, [SEPAL_LENGTH, SEPAL_WIDTH,
                                            PETAL_LENGTH, PETAL_WIDTH, CLASS])
        return list(reader)


def read_without_csv(filename):
    """Reads the iris file without using the csv module.

    Args:
        filename: The filename.

    Returns:
        A list of dictionaries mapping the features to their values.
    """
    data = []
    with open(filename, 'r') as iris_file:
        for line in iris_file.read().splitlines()[:-1]:
            features = line.split(',')
            data.append({SEPAL_LENGTH: features[0], SEPAL_WIDTH: features[1],
                         PETAL_LENGTH: features[2], PETAL_WIDTH: features[3],
                         CLASS: features[4]})
    return data


def make_data_numeric(data, *features):
    """For each entry in data, all features in features are cast to float.

    Args:
        data: The dataset to modify.
        features: The features to cast to float.
    """
    for date in data:
        for feature in features:
            date[feature] = float(date[feature])
    return data


def count_occurences(data, feature):
    """Counts how often each unique value of a feature occurs in the whole
    dataset.

    Args:
        data: The dataset to count.
```

```
        feature: The feature to count.

    Returns:
        A dictionary where the keys are the data values and the values are the
        occurences.
    """
    values = [d[feature] for d in data]
    return {k: values.count(k) for k in set(values)}


def mean(values):
    """Returns the mean of the values.

    Args:
        Values: A list of values.

    Returns:
        The mean.
    """
    return sum(values) / len(values)


def median(values):
    """Returns the median of the values.

    Args:
        Values: A list of values.

    Returns:
        The median.
    """
    idx = len(values) // 2
    if len(values) & 1:
        return sorted(values)[idx]
    else:
        return mean(sorted(values)[idx - 1:idx + 1])


def mode(values):
    """Returns the mode of the values.

    If multiples values tie, one value is returned.

    Args:
        values: A list of values.
```

```
    Returns:
        The mode.
    """
    counts = {k: values.count(k) for k in set(values)}
    return sorted(counts, key=counts.__getitem__)[-1]


def test():
    """Tests the statistical functions.

    Raises:
        AssertionError if a test fails.
    """
    testlist0 = [1, 2, 3, 4, 5]
    testlist1 = [1, 2, 3, 4, 5, 6]
    testlist2 = [2, 2, 3, 4, 4, 6]
    testlist3 = [2, 2, 3, 4, 5, 6, 7]

    assert mean(testlist0) - 5 <= 1e-6, mean(testlist0)
    assert mean(testlist1) - 3.5 <= 1e-6, mean(testlist1)
    assert mean(testlist2) - 21 / 6 <= 1e-6, mean(testlist2)
    assert mean(testlist3) - 29 / 7 <= 1e-6, mean(testlist3)

    assert median(testlist0) == 3, median(testlist0)
    assert median(testlist1) - 3.5 <= 1e-6, median(testlist1)
    assert median(testlist2) - 3.5 <= 1e-6, median(testlist2)
    assert median(testlist3) == 4, median(testlist3)

    assert mode(testlist3) == 2, mode(testlist3)


def main(with_csv=False):
    """Performs some simple data analysis.

    If with_csv is True, the csv module is used for loading the data.
    Otherwise, a simple custom solution is used.

    Args:
        with_csv: If True, uses the csv module.
    """
    if with_csv:
        data = read_with_csv(IRIS_FILE)
    else:
        data = read_without_csv(IRIS_FILE)
```

```
    data = make_data_numeric(data, SEPAL_LENGTH, SEPAL_WIDTH,
                             PETAL_LENGTH, PETAL_WIDTH)

    print('Total number of rows:', len(data))

    class_counts = count_occurences(data, CLASS)
    print('Instances:', class_counts)

    sepal_lengths = [d[SEPAL_LENGTH] for d in data]
    print('Mean sepal length (statistics):', statistics.mean(sepal_lengths))
    print('Mean sepal length (custom):', mean(sepal_lengths))

    sepal_l_setosa = [d[SEPAL_LENGTH] for d in data if 'setosa' in d[CLASS]]
    print('Mean sepal length (setosa, statistics):',
          statistics.mean(sepal_l_setosa))
    print('Mean sepal length (setosa, custom):', mean(sepal_l_setosa))

    sepal_widths = [d[SEPAL_WIDTH] for d in data]
    print('Median sepal width (statistics):', statistics.median(sepal_widths))
    print('Median sepal width (custom):', median(sepal_widths))

    sepal_w_virginica = [d[SEPAL_WIDTH] for d in data if 'vir' in d[CLASS]]
    print('Median sepal width (virginica, statistics):',
          statistics.median(sepal_w_virginica))
    print('Median sepal width (virginica, custom):', median(sepal_w_virginica))

    petal_l_versicolor = [d[PETAL_LENGTH] for d in data if 'ver' in d[CLASS]]
    print('Mode petal length (versicolor, statistics):',
          statistics.mode(petal_l_versicolor))
    print('Mode petal length (versicolor, custom):', mode(petal_l_versicolor))


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

Output:

```
Total number of rows: 150
Instances: {'Iris-setosa': 50, 'Iris-versicolor': 50, 'Iris-virginica': 50}
Mean sepal length (statistics): 5.843333333333334
Mean sepal length (custom): 5.843333333333335
Mean sepal length (setosa, statistics): 5.006
Mean sepal length (setosa, custom): 5.005999999999999
Median sepal width (statistics): 3.0
Median sepal width (custom): 3.0
```

```
Median sepal width (virginica, statistics): 3.0
Median sepal width (virginica, custom): 3.0
Mode petal length (versicolor, statistics): 4.5
Mode petal length (versicolor, custom): 4.5
```