PYTHON PROGRAMMING COOKBOOK

Hot Recipes for Python Development

SEBASTIAN VINCI

WEB CODE GEEKS
WEB DEVELOPERS RESOURCE CENTER

Python Programming Cookbook

Contents

Preface

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library. (Source: https://en.wikipedia.org/wiki/Python_%28programming_language%29)

In this ebook, we provide a compilation of Python examples that will help you kick-start your own projects. We cover a wide range of topics, from multi-threaded programming to web development with Django. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

Sebastian is a full stack programmer, who has strong experience in Java and Scala enterprise web applications.

He is currently studying Computers Science in UBA (University of Buenos Aires) and working a full time job at a .com company as a Semi-Senior developer, involving architectural design, implementation and monitoring.

He also worked in automating processes (such as data base backups, building, deploying and monitoring applications).

Chapter 1

CSV Reader / Writer Example

CSV (comma-separated values) is a standard for exporting and importing data across multiple formats, such as MySQL and excel.

It stores numbers and text in plain text. Each row of the file is a data record and every record consists of one or more fields which values are separated by commas. The use of the comma to separate every record's fields is the source of the name given to this standard.

Even with this **very** explicit name, there is no official standard CSVs, and it may denote some very similar delimiter-separated values, which use a variety of field delimiters (such as spaces and tabs, which are both very popular), and are given the .csv extension anyway. Such lack of strict terminology makes data exchange very difficult some times.

RFC 4180 provides some rules to this format:

- It's plain text
- Consists of records
- Every record consists of fields separated by a single character delimiter
- Every record has the same sequence of fields

But unless there is additional information about the provided file (such as if the rules provided by RFC were followed), data exchange through this format can be pretty annoying.

1.1 The Basics

Python has native support for CSV readers, and it's configurable (which, as we've seen, is necessary). There is a module csv which holds everything you need to make a CSV reader/writer, and it follows RFC standards (unless your configuration overrides them), so by default it should read and write valid CSV files. So, let's see how it works:

csv-reader.py

```
import csv
with open('my.csv', 'r+', newline='') as csv_file:
    reader = csv.reader(csv_file)
    for row in reader:
        print(str(row))
```

Here, we are importing csv and opening a file called *my.csv*, then we call csv.reader passing our file as a parameter and then we print each row in our reader.

If *my.csv* looks like this:

my.csv

```
my first column,my second column,my third column
my first column 2,my second column 2,my third column 2
```

Then, when you run this script, you will see the following output:

```
['my first column', 'my second column', 'my third column']
['my first column 2', 'my second column 2', 'my third column 2']
```

And writing is just as simple as reading:

csv-reader.py

```
import csv
rows = [['1', '2', '3'], ['4', '5', '6']]
with open('my.csv', 'w+', newline='') as csv_file:
    writer = csv.writer(csv_file)
    for row in rows:
        writer.writerow(row)

with open('my.csv', 'r+', newline='') as csv_file:
    reader = csv.reader(csv_file)
    for row in reader:
        print(str(row))
```

Then, in your csv file you'll see:

my.csv

```
1,2,3
4,5,6
```

And in your output:

```
['1', '2', '3']
['4', '5', '6']
```

It's pretty easy to see what is going on in here. We are opening a file in write mode, getting our writer from csv giving our file to it, and writing each row with it. Making it a little smarter:

csv-reader.py

```
import csv


def read(file_location):
    with open(file_location, 'r+', newline='') as csv_file:
        reader = csv.reader(csv_file)
        return [row for row in reader]


def write(file_location, rows):
    with open(file_location, 'w+', newline='') as csv_file:
        writer = csv.writer(csv_file)
        for row in rows:
            writer.writerow(row)


def raw_test():
    columns = int(input("How many columns do you want to write? "))
    input_rows = []
    keep_going = True
    while keep_going:
```

```
        input_rows.append([input("column {}: ".format(i + 1)) for i in range(0, columns)])
        ui_keep_going = input("continue? (y/N): ")
        if ui_keep_going != "y":
            keep_going = False

    print(str(input_rows))

    write('raw.csv', input_rows)
    written_value = read('raw.csv')
    print(str(written_value))

raw_test()
```

We ask the user how many columns does he want to write for each row and then ask him for a row as long as he wants to continue, then we print our raw input and write it to a file called raw.csv, then we read it again and print the data. When we run our script, the output will look like this:

```
How many columns do you want to write? 3
column 1: row 1, column 1
column 2: row 1, column 2
column 3: row 1, column 3
continue? (y/N): y
column 1: row 2, column 1
column 2: row 2, column 2
column 3: row 3, column 3
continue? (y/N):
[['row 1, column 1', 'row 1, column 2', 'row 1, column 3'], ['row 2, column 1', 'row 2,  ↩
    column 2', 'row 3, column 3']]
[['row 1, column 1', 'row 1, column 2', 'row 1, column 3'], ['row 2, column 1', 'row 2,  ↩
    column 2', 'row 3, column 3']]

Process finished with exit code 0
```

And, of course, our raw.csv looks like this:

raw.csv

```
"row 1, column 1","row 1, column 2","row 1, column 3"
"row 2, column 1","row 2, column 2","row 3, column 3"
```

Another rule the CSV format has, is the quote character. As you see, every input has a comma, which is our separator character, so the writer puts them between quoting marks (the default of the standard) to know that commas between them are not separators, but part of the column instead.

Now, although I would recommend leaving the configuration with its defaults, there are some cases where you need to change them, as you don't always have control over the csv's your data providers give you. So, I have to teach you how to do it (beware, great powers come with great responsibilities).

You can configure the delimiter and the quote character through `delimiter` and `quotechar` parameters, like this:

csv-reader.py

```
import csv


def read(file_location):
    with open(file_location, 'r+', newline='') as csv_file:
        reader = csv.reader(csv_file, delimiter=' ', quotechar='|')
        return [row for row in reader]


def write(file_location, rows):
    with open(file_location, 'w+', newline='') as csv_file:
```

```
        writer = csv.writer(csv_file, delimiter=' ', quotechar='|')
        for row in rows:
            writer.writerow(row)


def raw_test():
    columns = int(input("How many columns do you want to write? "))
    input_rows = []
    keep_going = True
    while keep_going:
        input_rows.append([input("column {}: ".format(i + 1)) for i in range(0, columns)])
        ui_keep_going = input("continue? (y/N): ")
        if ui_keep_going != "y":
            keep_going = False

    print(str(input_rows))

    write('raw.csv', input_rows)
    written_value = read('raw.csv')
    print(str(written_value))

raw_test()
```

So, now, having this console output:

```
How many columns do you want to write? 3
column 1: row 1 column 1
column 2: row 1 column 2
column 3: row 1 column 3
continue? (y/N): y
column 1: row 2 column 1
column 2: row 2 column 2
column 3: row 2 column 3
continue? (y/N):
[['row 1 column 1', 'row 1 column 2', 'row 1 column 3'], ['row 2 column 1', 'row 2 column 2 ↩
    ', 'row 2 column 3']]
[['row 1 column 1', 'row 1 column 2', 'row 1 column 3'], ['row 2 column 1', 'row 2 column 2 ↩
    ', 'row 2 column 3']]
```

Our raw.csv will like like this:

raw.csv

```
|row 1 column 1| |row 1 column 2| |row 1 column 3|
|row 2 column 1| |row 2 column 2| |row 2 column 3|
```

As you see, our new separator is the space character, and our quote character is pipe, which our writer is forced to use always as the space character is pretty common in almost every text data.

The writer's quoting strategy is also configurable, the values available are:

- `csv.QUOTE_ALL`: quotes every column, it doesn't matter if they contain a delimiter character or not.
- `csv.QUOTE_MINIMAL`: quotes only the columns which contains a delimiter character.
- `csv.QUOTE_NONNUMERIC`: quotes all non numeric columns.
- `csv.QUOTE_NONE`: quotes nothing. It forces you to check whether or not the user inputs a delimiter character in a column, if you don't, you will read an unexpected number of columns.

1.2 Reading and writing dictionaries

We've seen a very basic example of how to read and write data from a CSV file, but in real life, we don't want our CSV's to be so chaotic, we need them to give us information about what meaning has each of the columns.

Also, en real life we don't usually have our data in arrays, we have business models and we need them to be very descriptive. We usually use dictionaries for this purpose, and python gives us the tools to write and read dictionaries from CSV files.

It looks like this:

```
import csv

dictionaries = [{'age': '30', 'name': 'John', 'last_name': 'Doe'}, {'age': '30', 'name': ' ↩
    Jane', 'last_name': 'Doe'}]
with open('my.csv', 'w+') as csv_file:
    headers = [k for k in dictionaries[0]]
    writer = csv.DictWriter(csv_file, fieldnames=headers)
    writer.writeheader()
    for dictionary in dictionaries:
        writer.writerow(dictionary)

with open('my.csv', 'r+') as csv_file:
    reader = csv.DictReader(csv_file)
    print(str([row for row in reader]))
```

We are initializing a variable called `dictionaries` with an array of test data, then we open a file in write mode, we collect the keys of our dictionary and get a writer of our file with the headers. The first thing we do is write our headers, and then write a row for every dictionary in our array.

Then we open the same file in read mode, get a reader of that file and print the array of data. You will see an output like:

```
[{'name': 'John', 'age': '30', 'last_name': 'Doe'}, {'name': 'Jane', 'age': '30', ' ↩
    last_name': 'Doe'}]
```

And our csv file will look like:

my.csv

```
name,last_name,age
John,Doe,30
Jane,Doe,30
```

Now it looks better. The CSV file has our headers information, and each row has the ordered sequence of our data. Notice that we give the file names to our writer, as dictionaries in python are not ordered so the writer needs that information to write each row with the same order.

The same parameters apply for the delimiter and the quote character as the default reader and writer.

So, then again, we make it a little smarter:

csv-reader.py

```
import csv

def read_dict(file_location):
    with open(file_location, 'r+') as csv_file:
        reader = csv.DictReader(csv_file)
        print(str([row for row in reader]))
        return [row for row in reader]


def write_dict(file_location, dictionaries):
    with open(file_location, 'w+') as csv_file:
```

```
        headers = [k for k in dictionaries[0]]
        writer = csv.DictWriter(csv_file, fieldnames=headers)
        writer.writeheader()
        for dictionary in dictionaries:
            writer.writerow(dictionary)


def dict_test():
    input_rows = []
    keep_going = True
    while keep_going:
        name = input("Name: ")
        last_name = input("Last Name: ")
        age = input("Age: ")
        input_rows.append({"name": name, "last_name": last_name, "age": age})
        ui_keep_going = input("continue? (y/N): ")
        if ui_keep_going != "y":
            keep_going = False

    print(str(input_rows))

    write_dict('dict.csv', input_rows)
    written_value = read_dict('dict.csv')
    print(str(written_value))

dict_test()
```

And now when we run this script, we'll see the output:

```
Name: John
Last Name: Doe
Age: 30
continue? (y/N): y
Name: Jane
Last Name: Doe
Age: 40
continue? (y/N):
[{'age': '30', 'last_name': 'Doe', 'name': 'John'}, {'age': '40', 'last_name': 'Doe', 'name ↩
    ': 'Jane'}]
[{'age': '30', 'last_name': 'Doe', 'name': 'John'}, {'age': '40', 'last_name': 'Doe', 'name ↩
    ': 'Jane'}]
```

And our dict.csv will look like:

csv-reader.py

```
age,last_name,name
30,Doe,John
40,Doe,Jane
```

A little side note: As I said before, dictionaries in python are not ordered, so when you extract the keys from one to write its data to a CSV file you should order them to have your columns ordered always the same way, as you do not know which technology will your client use to read them, and, of course, in real life CSV files are incremental, so you are always adding lines to them, not overriding them. Avoid any trouble making sure your CSV will always look the same.

1.3 Download the Code Project

This was an example on how to read and write data from/to a CSV file.

Download You can download the full source code of this example here: **python-csv-reader**

Chapter 2

Decorator Tutorial

Sometimes, we encounter problems that require us to extend the behavior of a function, but we don't want to change its implementation.Some of those problems could be: logging spent time, caching, validating parameters, etc. All these solutions are often needed in more than one function: you often need to log the spent time of every http connection; you often need to cache more than one data base entity; you often need validation in more than one function. Today we will solve these 3 mentioned problems with Python decorators:

- Spent Time Logging: We'll decorate a couple functions to tell us how much time do they take to execute.
- Caching: We'll add cache to prevent a function from executing when its called several times with the same parameters.
- Validating: We'll validate a function's input to prevent run time errors.

2.1 Understanding Functions

Before we can jump into decorators, we need to understand how functions actually work. In essence, functions are procedures that return a value based on some given arguments.

```
def my_function(my_arg):
    return my_arg + 1
```

In Python, functions are first-class objects. This means that functions can be assigned to a variable:

```
def my_function(foo):
    return foo + 1
my_var = my_function
print(str(my_var(1))) # prints "2"
```

They can be defined inside another functions:

```
def my_function(foo):
    def my_inner_function():
        return 1
    return foo + my_inner_function()
print(str(my_function(1))) # still prints "2"
```

They can be passed as parameters (higher-order functions):

```
def my_function(foo, my_parameter_function):
    return foo + my_parameter_function()
def parameter_function(): return 1
print(str(my_function(1, parameter_function))) # still prints "2"
```