Python and Python Web Applications

ICT Enterprise Project

Supervisor: Catherine Woods

Submitted By: David Murphy (TL\_KCOMP\_GY6)

<<Submitted on: 28th April 2014>>

Table of Contents

[1 Introduction 1](#_Toc382239365)

[2 Requirements Specification 2](#_Toc382239366)

[2.1 Project goals 2](#_Toc382239367)

[3 Project Plan 3](#_Toc382239368)

[4 Python 5](#_Toc382239369)

[4.1 Introduction to Python 5](#_Toc382239370)

[4.2 History of Python 6](#_Toc382239371)

[4.3 Python Philosophy 7](#_Toc382239372)

[4.4 Overview of Python Features 8](#_Toc382239373)

[4.4.1 Interpreted Language 8](#_Toc382239374)

[4.4.2 Multi-Paradigm 8](#_Toc382239375)

[4.4.3 Cross-Platform Usage 8](#_Toc382239376)

[4.4.4 Extensibility and Libraries 9](#_Toc382239377)

[4.4.5 Third-Party Support 9](#_Toc382239378)

[4.5 Python Syntax and Semantics 10](#_Toc382239379)

[4.5.1 Comments 10](#_Toc382239380)

[4.5.2 Number Operations 10](#_Toc382239381)

[4.5.3 Strings 10](#_Toc382239382)

[4.5.4 Boolean operators 11](#_Toc382239383)

[4.5.5 Comparative operators 11](#_Toc382239384)

[4.5.6 Membership Operators 12](#_Toc382239385)

[4.5.7 Numeric data types 12](#_Toc382239386)

[4.5.8 Sequence types 12](#_Toc382239387)

[4.5.9 Variable usage 14](#_Toc382239388)

[4.6 Control Flow 16](#_Toc382239389)

[4.6.1 Indentation and variable scope 16](#_Toc382239390)

[4.6.2 if statements 16](#_Toc382239391)

[4.6.3 for loop 17](#_Toc382239392)

[4.6.4 while loop 18](#_Toc382239393)

[4.7 Functions 19](#_Toc382239394)

[4.7.1 Functions 19](#_Toc382239395)

[4.7.2 Methods 19](#_Toc382239396)

[4.7.3 Default Argument Values 20](#_Toc382239397)

[4.7.4 Keyword Arguments 20](#_Toc382239398)

[4.7.5 Returning values from functions 21](#_Toc382239399)

[4.7.6 Lambda functions 22](#_Toc382239400)

[4.7.7 Variable numbers of arguments to a function 22](#_Toc382239401)

[4.8 Exception Handling 23](#_Toc382239402)

[4.8.1 Try...except construct 23](#_Toc382239403)

[4.8.2 Exception types 24](#_Toc382239404)

[4.8.3 Raising exceptions and exception arguments 24](#_Toc382239405)

[4.9 Object-Oriented Programming 26](#_Toc382239406)

[4.10 Libraries 27](#_Toc382239407)

[5 Python Web Framework 28](#_Toc382239408)

[5.1 Introduction 28](#_Toc382239409)

[5.2 Python Web Frameworks 28](#_Toc382239410)

[5.3 Flask 28](#_Toc382239411)

[6 Comparison of Python with Java 29](#_Toc382239412)

[7 Heroku Cloud Application Platform 30](#_Toc382239413)

[7.1 Introduction to Heroku 30](#_Toc382239414)

[7.2 Heroku Applications 30](#_Toc382239415)

[7.2.1 Web Servers 30](#_Toc382239416)

[7.2.2 Deploying to Heroku with Git 30](#_Toc382239417)

[7.2.3 Managing Heroku Applications 31](#_Toc382239418)

[7.3 Use of Heroku with Project Implementation 32](#_Toc382239419)

[8 Python/Flask Web Application 33](#_Toc382239420)

[8.1 Introduction 33](#_Toc382239421)

[8.2 Technologies 33](#_Toc382239422)

[8.3 Product Delivery 34](#_Toc382239423)

[8.4 Analysis 35](#_Toc382239424)

[8.4.1 Requirements Hierarchy Chart 35](#_Toc382239425)

[8.4.2 User Requirements 36](#_Toc382239426)

[8.4.3 System Requirements 36](#_Toc382239427)

[8.5 Design 36](#_Toc382239428)

[8.5.1 Initial Design Development 36](#_Toc382239429)

[8.5.2 Implementation 38](#_Toc382239430)

[8.6 Testing Strategy 38](#_Toc382239431)

[8.7 Implementation Increments 39](#_Toc382239432)

[8.7.1 Increment #01 39](#_Toc382239433)

[8.7.2 Increment #02 45](#_Toc382239434)

[8.7.3 Increment #03 54](#_Toc382239435)

[8.7.4 Increment #04 57](#_Toc382239436)

[8.7.5 Increment #05 63](#_Toc382239437)

[8.8 Final Implementation 74](#_Toc382239438)

[8.9 Implementation Conclusions 75](#_Toc382239439)

[9 Project Conclusions 76](#_Toc382239440)

[10 References 77](#_Toc382239441)

[11 Bibliography 78](#_Toc382239442)

REFERENCING – for citing

BIBLIOGRAPHY – other resources not cited

# Introduction

Introduction goes here.

# Requirements Specification

## Project goals

The goals of this project are as follows:

* Acquire programming skills in the Python programming language
* Acquire programming skills in a Python web framework (Flask)
* Compare and contrast Python with another high-level programming language (Java) in terms of programming paradigms, syntax, standard libraries, domain applications and web development
* Perform analysis, design, implementation and testing of a web application programmed in a Python/Flask environment

# Project Plan

The following is a detailed breakdown of the tasks required to achieve the project goals outlined above:

|  |  |
| --- | --- |
| **PROJECT GOALS** | **TIMELINE - Wk1 – Wk12 (college term)** |
| **Acquire Python programming skills**  Tasks to complete:   * study official Python.org documentation * complete official Python.org tutorial * complete ‘Dive Into Python’ online course   All research conducted in this phase will be documented in this section accordingly. | This research will be carried out continuously over Wks1 – Wks6 |
| **Acquire skills in a Python web framework (Flask)**  Tasks to complete:   * read Flask official documentation * complete Flask online tutorial * complete other online tutorials   All research conducted in this phase will be documented in this section accordingly. | This research will be carried out continuously over Wks3 – Wks7 |
| **Comparison of Python with another high-level programming language (Java)**  A critical comparison of Python with Java in terms of the following headings will be presented:   * programming paradigms * features of the language * syntax and semantics * standard libraries * domain applications * web development | Work in this section will be carried out over Wk5 – Wk8 |
| **Analysis, design, implementation & testing of a web application programmed in a Python/Flask environment**  Summary of implementation  The application to be developed will allow a client-side user to log on to a web application, select from a list of currencies and input a monetary amount to be converted from one denomination to another.  When instructed, the web application will connect RESTfully to a remotely hosted web service using the required input data. The web service will perform the necessary business logic calculation and RESTfully return data to the web application for presentation to the user.  The web service could be enhanced by connecting to a database to retrieve currency data.  Technologies to be employed   * The web application and web service will be developed in Python using the Flask web micro-framework for web development. * The web service will be remotely hosted on a suitable hosting service, such as Heroku * Analysis and design of the application will include analysis of the problem and a description of the solution design, and will also include UML diagrams as required.   Product delivery  Due to the nature of the research into Python and Flask being carried out the application will be developed incrementally, with each increment adding further functionality and/or improvements to the application until a satisfactory solution is implemented.  Testing strategy  A testing strategy will be designed and the application tested accordingly in conjunction with each increment. | Work in this section will be carried out over Wk3 – Wk11  Analysis & design to be completed by Wk4  First increment delivered Wk4  Weekly increments thereafter until satisfactory implementation is achieved  Testing strategy to be completed by Wk4, and carried out in conjunction with the various increments  Final submission 28/04/2014 |

# Python

## Introduction to Python

Python is a powerful, high-level and dynamic programming language. It is an interpreted object-oriented language that is available across many platforms. As a language it is extensible, allowing users to add to the language with their own source code, modules and components which can be reused in other applications.[[1]](#footnote-2)

Python supports multiple programming paradigms including object-oriented, procedural and functional styles.[[2]](#footnote-3) It is very flexible in its usage in that it is suitable for use in large-scale application development, or equally well as a scripting language.

Python promotes itself as being a very powerful, high-level language while being expressive and easy to learn. Some of its features include:[[3]](#footnote-4)

* clear, readable syntax
* intuitive object orientation
* natural expression of procedural code
* full modularity
* very high-level dynamic data types
* comprehensive standard library

Python has a vast array of domain applications, notably web development, database-driven development, GUI development, scientific applications, education, general software development and game development.[[4]](#footnote-5)

Python is developed under an approved open-source license meaning that the language is free for all to download and use. The intellectual property rights behind the Python language are owned by the Python Software Foundation. [[5]](#footnote-6) This Foundation forms the core of a wider community of developers and contributors that maintains and continually develops the language.[[6]](#footnote-7)

## History of Python

Python first emerged in the late 1980’s when Dutch programmer, Guido van Rossum was working with a language called ABC on the Amoeba operating system, which at the time was an experimental distributed operating system. ABC had similarities to BASIC or Pascal; it was a useful teaching language, it did not require variable declarations and used indentation for nesting of statements. Van Rossum liked these aspects of ABC. However, other aspects of the language were not so positive; the language had poorly structured error handling, and was monolithic in nature such that it could not be easily extended or modified.[[7]](#footnote-8)

Van Rossum decided to take the best features of ABC and incorporate them into a new language called Python. Python was designed to be very readable, using English keywords where other languages might use punctuation, and adopting an uncluttered code layout. It had fewer syntactic constructions than other languages such as C, Perl or Pascal.7

Some years later, Van Rossum and his team of programmers worked in an open-structured programming laboratory, allowing the language to be developed in new and interesting ways. The team borrowed ideas and concepts from other programming languages, eventually releasing Python version 1.0 to the programming community in 1994. Python 2.0 was released in 2000, with notable introductions to this version being the use of list comprehensions (borrowed from the Haskell programming language) and a garbage collection system.[[8]](#footnote-9)

The introduction of Version 2.0 heralded the advent of a more open, transparent and community-backed development of the language.[[9]](#footnote-10)

Python is currently being developed in two strands, versions 2.x and 3.x. Version 2.7.6 is the current version 2.x, while version 3.3.4 is the latest version 3.x available at the time of writing. Version 3.x has broken backward compatibility with Versions 2.x and under, and Version 2.7 will be the last major version 2.x to be released.[[10]](#footnote-11)

Because the language was developed to be deliberately simple in structure and syntax, it is easy to learn and is suitable as a first language for beginners. In spite of this, it is a very powerful language that can accomplish impressive tasks with a minimum amount of coding.7

## Python Philosophy

Python was designed to be easily extensible, having a relatively small core language, with further extended functionality being added as required. It can also be embedded in existing applications that need a programmable interface. Python philosophy rejects complicated or complex code and exuberant syntax in favour of a sparser, less-cluttered grammar. The language is intended to be fun to use, and this spirit is often reflected in the official documentation.[[11]](#footnote-12)

The endeavour towards simple code constructs, intuitive syntax and readable code has lead to the so-called *Zen of Python*, in which the language’s guiding principles have been packaged into 20 aphorisms, some of which are repeated here:[[12]](#footnote-13)

* Beautiful is better than ugly
* Explicit is better than implicit
* Simple is better than complex
* Complex is better than complicated
* Sparse is better than dense
* Readability counts
* Special cases aren’t special enough to break the rules
* There should be one—and preferably only one –obvious way to do it

## Overview of Python Features

### Interpreted Language

Python is an interpreted language. Rather than compiling the source code into machine code and executing that, Python source code is loaded directly into the Python interpreter and executed there. The interpreter converts the source code into bytecode but does not process the file at this point. Once the user requests that the interpreted code be executed, the interpreter begins to read each line of bytecode and validates it. It is at this run-time stage that any syntax errors will become apparent (as opposed to a compiled language, syntax errors become apparent at compile-time).[[13]](#footnote-14)

Python provides an interactive command-line interpreter, at which a user can enter Python commands and statements directly to be executed in real-time.

### Multi-Paradigm

Python supports various programming paradigms:

* Object-oriented – Python includes such object-oriented concepts as class definition defining object state and behaviour, data members, instance variables, function overloading, instantiation of objects, inheritance and aggregation. In fact, everything in Python is an object.[[14]](#footnote-15) Extensibility and code reuse are natural by-products of the object-oriented paradigm. Python does not force the use of object-oriented features.[[15]](#footnote-16)
* Procedural – Python is useful as a procedural language. Lists of instructions can be quickly drafted and executed. Python is widely used for scripting purposes.
* Functional – Python supports functional programming, in which functions strive for side-effect free behaviour. Advantages of this programming style include formal provability, modularity, composability, and ease of debugging and testing.15

### Cross-Platform Usage

The Python interpreter has been developed for many operating platforms including Windows, Linux and OSX. A script written in Python on any platform will happily execute on another supported platform without any alterations to the code; the code in a Python program is the same no matter what the underlying architecture. The concept of “write once, run anywhere” is justifiably applicable to Python.

Python itself is written in C, and the maintained version of the language engine itself is known as CPython. Other implementations of Python exist, such as Jython (which is implemented in Java and compiles to Java bytecode) and IronPython (which runs on Microsoft’s CLR).

### Extensibility and Libraries

The modularity provided by object-oriented concepts allows for easy extension of functionality of the language. This is evidenced by the large array of frameworks developed by third parties for providing particular functionalities, in particular open-source web frameworks such as Django, Bottle and Flask for web application development.

There are many general-purpose libraries included with the language, for which the source code is available, that can be easily reused as required.

### Third-Party Support

Python is well supported in the developer community by such third-party tools as MySQL, an open-source database package, and Apache, an open-source web-server. Python can also be easily embedded in other C/C++ applications.

## Python Syntax and Semantics

As indicated by the various aphorisms in the *Zen of Python* language philosophy (see Section 4.3), Python was designed to be a very readable language. It has an uncluttered visual layout, using English keywords where other languages may use punctuation.[[16]](#footnote-17) There follows an introduction to the syntax and semantics of the language (all information in this section is adapted from the official Python documentation, located at <http://docs.python.org/2/>).

Note that all reference to Python code and code provided as part of this project refer to Python Version 2.7.6. This version is chosen over Version 3.3.4 due to the fact that Version 2.7 currently has wider usage in the Python community in general, and therefore better third-party support. While this is likely to change in the future, it is considered appropriate that, for this project, Version 2.7.6 is used.

### Comments

Single-line comments are written by commencing the line with the hash (#) character. The end-of-line indicates the end of the comment.

For example:

1 # this is a comment in Python

2

3 # this is another comment

### Number Operations

Binary operators + - \* / and % (modulo) operate in the same mathematical manner as other languages. Some points should be noted regarding numbers and associated operations:

* Integer division returns a floor value. For example 7/3 returns 2.
* If one of the arguments in division is a float, then the operation returns a float. For example, 7.0/3 returns 2.3333...
* \*\* operator returns an exponential power. For example, 2\*\*5 returns 32.
* Complex numbers are supported using the suffix ‘j’ or ‘J’, as in (real + imag j), where j is the square root of -1, and ‘real’ and ‘imag’ are the real and imaginary components of the vector respectively.

### Strings

In Python, a string is a *Sequence Type.* String objects can be enclosed in single or double quotes (as in ‘String’ or “String”). Quote characters required in strings can be escaped with the ‘\’ character. Other escape sequences include \n for new line, \t for horizontal tab and \v for vertical tab. Individual strings can span multiple lines if surrounded by triple quotes (‘’’ or “””).

Strings can be concatenated with the ‘+’ operator, and repeated with the ‘\*’ operator. Strings immediately next to each other are automatically concatenated. For example, ‘ABC’ ‘xyz’ yields ‘ABCxyz’.

There is no ‘character’ or ‘char’ data type in Python. A single character is represented by a string of length 1. However, characters in a string are indexed, and can be accessed using ‘slice notation’, with the first character at index 0. For example:

x = Python

x[0] will return ‘P’

x[2] will return ‘t’

x[2:4] will return ‘ty’

x[3:] will return ‘hon’ (ie, the remainder of the string from index 3)

x[:3] will return ‘Pyt’ (ie, the string as far as index 3)

Negative indices are counted from the right-hand-side, starting at -1. Therefore:

x[-1] will return ‘n’

x[-5] will return ‘y’

It is important to note that strings are immutable. In other words, x[0] = ‘W’ would return an error. Placing a ‘u’ character before a string literal encodes the string as Unicode, for example: u’Python’.

### Boolean operators

Python includes Boolean operators as follows:

|  |  |
| --- | --- |
| Operator | Result |
| x or y | If x is false, then y, else x |
| x and y | If x is false, then x, else y |
| not x | If x is false, then True, else False |

### Comparative operators

Python provides for object identity comparison with the ‘is’ and ‘is not’ operators.

Logical operators <, >, <=, >=, ==, != are similar to those used in mathematics and other languages.

### Membership Operators

Python provides membership operators. The operator ‘in’ indicates whether a given sequence contains a value. The operator ‘not in’ indicates whether a given sequence does not contain a given value.

### Numeric data types

There are four distinct numeric types in Python; plain integers (int), long integers (long), floating point numbers (float) and complex numbers (complex). Plain integers have 32-bit precision, long integers have unlimited precision. Floats are usually equivalent to a *double* in C, and precision can vary depending on the host machine. Both real and imaginary parts of a complex number are floating point values.

There are additional numeric types included in the Python standard library, namely fraction, which holds rational numbers, and decimal, which holds floating point numbers with user-definable precision.

The built-in numeric types support various operations such as:

abs (x) returns the magnitude of x

int (x) casts value of x to an int (similar for long and float)

pow (x, y) returns x to the power of y

### Sequence types

There are various sequence types in Python, among the most common of which are strings, lists and tuples.

#### Lists

The most versatile sequence type is the list. A list is a set of comma separated values surrounded by square brackets. Items in a list do not have to be of the same data type. Similarly to strings, list item indices start at 0, and can be sliced and concatenated. For example:

mylist = [‘dog’, ‘cat’, 1234, 500]

mylist[3].... returns 500

mylist[-2].... returns 1234

mylist[1:-1].... returns [‘cat’, 1234]

mylist[:2] + [‘giraffe’, 3\*4].... returns [‘dog’, ‘cat’,

‘giraffe’, 12]

mylist[:].... returns [‘dog’, ‘cat’, 1234, 500]

It is possible to work with values of individual items in a list. For example:

mylist[3] = mylist[3] + 250 returns [‘dog’, ‘cat’, 1234, 750]

Items can be replaced, inserted or removed as follows:

mylist[0:2] = [1, 10].... returns [1, 10, 1234, 750]

mylist[0:2] = [].... returns [1234, 750]

mylist[1:1] = [‘xyz’, ‘pqr’].... returns [1234, ‘xyz’, ‘pqr’,

750]

List length can be determined using the len() function:

len(mylist).... returns 4

Lists can be nested. For example:

q = [2,3]

p = [1, q, 4].... returns [1, [2, 3], 4]

Items can be appended to lists with the append() method:

p.append(5).... returns [1, [2, 3], 4, 5]

q.append(‘xtra’) .... returns [1, [2, 3, ‘xtra’], 4, 5]

#### Tuples

A tuple is a series of comma separated values, usually surrounded by parenthesis (although parenthesis are not required if the tuple assignment is the only statement on that line of code). Tuples are immutable, though it is possible to assign values to tuples which are mutable, such as lists.

Examples of tuples include:

tuple1 = (a, b, c) tuple of variable names

tuple2 = (‘a’, ‘b’, ‘c’) tuple of strings

tuple3 = (1, 2, [‘a’, ‘b’, ‘c’], 3) tuple of numbers and a list

#### Dictionaries

A dictionary is an unordered collection of key-value pairs. The key can be any immutable data type, such as a strings, numbers or tuples containing only immutable types. Lists cannot be used as keys. Keys must be unique within a dictionary, and each key references a value. The value can be any data type, including strings, numbers, lists, tuples or other dictionaries.

Dictionaries are created using curly braces {}. A pair of curly braces {} on its own will create an empty dictionary. Dictionary data can be initialised by inserting key:value pairs separated by commas within the braces, as follows:

dict1 = {‘id1’: 1000, ‘id2’: 1001, ‘id3’: 1002}

dict2 = {‘person1’: ‘John’, ‘person2’: ‘Mary’}

dict3 = {‘person2’: {‘name’: ‘Jack’, ‘age’: 23},

‘person3’: [‘Jane’, 35]}

In the third dictionary, dict3, the value in the first key-value pair is another dictionary, while the value in the second key-value pair is a list.

The main operations associated with a dictionary are storing a value with an associated key, and retrieving a value associated with a particular key. Using the above examples, the following show how to manipulate dictionaries:

dict1[‘id4’] = 1003 #insert a value to dict1

print dict1....{‘id1’: 1000, ‘id2’:1001, ‘id3’: 1002,

‘id4’: 1003}

print dict1[‘id1’] #get a value with key ‘id1’

....1000

The keys() method of a dictionary object returns a list of all the keys used in that dictionary in arbitrary order:

print dict1.keys()

....[‘id1’, ‘id2’, ‘id3’, ‘id4’]

Dictionaries can also be created using the dict() constructor, which takes as an argument a sequence of key-value pairs in tuple format.

### Variable usage

Python is not a “strongly typed language”, in that variables do not have specific types which they can only be assigned to for the life of the execution.[[17]](#footnote-18) Variables do not have to be declared; a variable exists as soon as its name is referred to in the code. Variables are not assigned to a type, so a variable name can happily refer to different object types throughout its scope.

Naming convention in Python dictates that variable names should be all in lower case, with individual words separated by the underscore (\_) character. It must begin with a letter or underscore, and the name is case-sensitive. Values are assigned with the assignment (=) operator.

Some examples:

x = 23

y = x + 4

print y.... returns 27

x = ‘hello’

y = ‘ there’

print x + y.... returns ‘hello there’

In this example, the variable x and y are assigned to integers initially, and then to strings without causing any difficulty.

Python caters for multiple assignment, as in the following example where a and b are assigned the values 1 and 2:

a, b = 1, 2

Variables can be defined as global with the ‘global’ keyword. Global variables have scope throughout the code.

## Control Flow

### Indentation and variable scope

In Python, code blocks are controlled using indentation. There are no curly braces ({}) required to define code blocks. End-of-line is determined by carriage returns, rather than semi-colons found in other languages.

The following is a simple example:

x = 10

if x == 10:

y = ‘y has local scope within indented block’

print y

This will print the value of y (‘y has local scope within indented block’).

However, the following will generate an error:

x = 10

if x == 10:

print x

y = ‘y has no scope outside the indented block’

print y.... error

y has local scope only and cannot be used unassigned outside the indented block.

### if statements

If-statements must be followed immediately by a colon (:) and an indented code block. elif is a keyword used for else-if. For example:

x = 4

if x < 0:

print ‘x is negative’

elif x == 0:

print ‘x is zero’

else:

print ‘x is positive’

The above example obviously prints ‘x is positive’.

### for loop

Unlike other languages, the Python for-statement iterates over the items of any sequence types (such as a list or string) in the order that they appear in the sequence.

For example:

languages = [‘Python’, ‘Java’, ‘Ruby’]

for my\_item in languages:

print my\_item, len(my\_item)

This will return the following output:

Python 6

Java 4

Ruby 4

If a sequence needs to be modified as a for loop iterates over it, slice notation can be used to create a copy of the sequence for iteration, as follows:

for my\_item in languages[:]:

if len(my\_item) > 5:

languages.insert(0, my\_item)

print my\_item

Here the for-loop iterates over a copy of languages, and inserts any item with length > 5 at index 0, as follows:

[‘Python’, ‘Python’, ‘Java’, ‘Ruby’]

If a for-loop is needed over an arithmetic progression, the range() function can be used:

range(6).... returns [0, 1, 2, 3, 4, 5]

range(5, 10).... returns [5, 6, 7, 8, 9]

range (0, 10, 3).... returns [0, 3, 6, 9]

Combined with a for-loop:

for i in range(10):

print i

The break and continue statements can be used to break out of a loop and continue with the next iteration respectively.

### while loop

The while-loop is executed similarly to the for-loop, as follows:

a = 1

while a < 10:

print a,

a = a + 1

The comma in the print statement prevents the newline return. The following is the output expected:

1 2 3 4 5 6 7 8 9

## Functions

### Functions

Functions are defined in Python with the ‘def’ keyword, an end-of-statement colon (:) and indentation to define the code block. For example, the following function definition to print a Fibonacci series from the python.org tutorial nicely demonstrates a function definition:

def fib(n):

a, b = 0, 1

while a < n:

print a,

a, b = b, a + b

A sample call of the function, fib(50) will print the following output:

0 1 1 2 3 5 8 13 21 34

‘Docstrings’ can be included as the first statement of a function, and must be a string literal describing the function surrounded by triple-quotes (‘’’docstring’’’). This docstring is used for automatically generating documentation (similar to Javadoc in Java).

Global variables cannot be directly assigned a value within a function, but they can be referenced.

Arguments are passed to a function using the ‘call-by-value’ method. This means that the actual parameter passed to the formal parameter in the function is always an *object reference* rather than the actual value of the object.

A function definition creates a new entry in the program *symbol table*, which is a table of all variables used in the program. Therefore, a function name is essentially a variable name, and can be assigned to a new name as required. For example, taking the fib() function above:

new\_fib = fib

Now, when the functinon new\_fib(50) is called we get the same output as when we called fib(50).

The ‘return’ keyword returns a value similar to other languages.

### Methods

In Python, a *method* is a function that ‘belongs’ to an object, and is called using dot notation, as in obj.methodname(). For example, if we have a list object ‘my\_list’, we can call the append() method as follows:

my\_list.append(‘an example’)

which appends the argument to the list.

### Default Argument Values

In function definition, default values can be specified for one or more formal parameters. This allows the function to be called using some or all of these arguments. When arguments are not passed, the default value in the formal parameter is assigned to the argument. Mandatory arguments must appear first in the parameter list.

For example, take the following function definition[[18]](#footnote-19):

Def ask\_ok(prompt, retries=4, complaint=’Yes or no, please’):

while True:

ok = raw\_input(prompt)

if ok in ('y', 'ye', 'yes'):

return True

if ok in ('n', 'no', 'nop', 'nope'):

return False

retries = retries - 1

if retries < 0:

raise IOError('refusenik user')

print complaint

This function can be called in several ways:

* Provide the mandatory argument (prompt) :

ask\_ok(‘Do you want to quit?’)

* Provide one of the optional arguments:

ask\_ok(‘Do you really want to quit?’, 3)

* Provide all the arguments:

ask\_ok(‘Overwrite file?’, 2, ‘Must be yes or no!’)

### Keyword Arguments

Python allows for the sending of a keyword name with an argument to a function. For example, take the following function:

def say\_hello(a=’hello’, b=’there’):

print a + b

Now, we can call the function in the following ways:

* say\_hello()

.... ‘hello there’

* say\_hello(a=’hi’)

.... ‘hi there’

* say\_hello(b=’hello’)

.... ‘hello hello’

* say\_hello(b=’again’, a=’hi’)

.... ‘hi again’

Keyword arguments must follow positional arguments.

Functions can also be passed as arguments to another function.

### Returning values from functions

Single values are returned from functions in Python using the return keyword, similar to other languages. In addition, Python functions can return multiple values of varying data types. For example, the following function will return the argument value, the value squared and the value cubed:

def multi\_return(x):

return x, x\*x, x\*x\*x

The following function will return multiple data types (numeric and string):

def another\_multi\_return(a,b):

a=21

b=’Python’

return a, b

What happens in reality within the function is that a tuple with the required returned values is created within the function, and then unpacked into its component parts on exiting the function.[[19]](#footnote-20) This is seamlessly carried out behind the scenes, and the values are neatly returned to the user individually.

### Lambda functions

Lambda functions can be thought of as small anonymous functions. Typically, a lambda function is implemented to do something very trivial[[20]](#footnote-21), such as multiplying a number or modifying a string. The syntax of a lambda function is as follows:

name = lambda (arguments) : code

The following are some simple examples of lambda functions:

* get\_sum = lambda (a, b) : a + b

get\_sum(2,3)

.... 5

* get\_square = lambda (a, b) : a \*\* b

get\_square(5, 2)

.... 25

### Variable numbers of arguments to a function

Python allows for the coding of a function that accepts a variable number of arguments. This is useful if the number of arguments at the time of function definition is unknown. A single asterisk ‘\*’ is placed in front of the parameter name in order to achieve this. Consider the following function:[[21]](#footnote-22)

def average( \*args ) :

total = 0

count = 0

for i in args :

total = total + i

count = count + 1

return total / count

The function calculates the average of all values passed to it. Thus, the following calls to the function can be made:

* average(1, 3)

.... 4

* average(1, 2, 3)

.... 2

* average(5, 4, 3, 2, 1)

.... 3

## Exception Handling

### Try...except construct

Exception handling is easily accomplished in Python by means of the try...except...else...finally construct, similar to try...catch constructs found in other languages.

The basic syntax of a try...except construct is as follows:[[22]](#footnote-23)

try:

# some python statements

except [ExceptionName]:

# some error handling if exception raised

else:

# some other statement if no exceptions are raised

finally:

# do something when all is done in any case

The try clause and one except clause are required as a minimum in an exception handler. Any number of except clauses can be included, and the else and finally clauses are optional.

The following example[[23]](#footnote-24) takes user input from the console using the raw\_input() function, and demonstrates the use of multiple except clauses:

try:

v = int( raw\_input("Enter a value: "))

print "We got some valid input!"

except ValueError:

print "Invalid input, please enter a value"

except KeyboardInterrupt:

print "Please don't hit ctrl-c"

Here, the ValueError and KeyboardInterrupt exceptions are handled separately. Alternatively, multiple exceptions can be handled in the same except clause:

try:

v = int( raw\_input("Enter a value: "))

print "We got some valid input!"

except (ValueError, KeyboardInterrupt):

print "Invalid input, or Keyboard Interrupt"

### Exception types

There is a number of standard exception types provided in Python. Some of these include:

* Exception the base for all other exception types in Python
* FloatingPointError error in processing floating point value
* MemoryError raised when system runs out of memory while running
* NameError raised when a name cannot be found
* ValueError value assigned to a given type is not valid
* ZeroDivisionError raised when division by zero occurs

User-defined exceptions can also be defined in Python.

### Raising exceptions and exception arguments

Exceptions are easily raised in Python using the raise keyword. For example, to raise a ValueError the following could be coded:[[24]](#footnote-25)

def myFunction( a, b, c ) :

if a < 0 or a > 10 :

raise ValueError, "a must be between 0 and 10"

if b > 50 :

raise ValueError, "b must be less than 50"

return a\*b + c

This simple example raises a ValueError if the various user-defined conditions are not met. The exception is also passed a single string argument with a useful user-defined error message which is presented to the user in a traceback if the exception is raised.

## Object-Oriented Programming

## Libraries

# Python Web Frameworks

## Introduction

## Python Web Frameworks

## Flask

# Comparison of Python with Java

# Heroku Cloud Application Platform

## Introduction to Heroku

Heroku is a “polyglot cloud application platform”.[[25]](#footnote-26) It is a cloud web application platform service that hosts web applications developed in various supported languages. The platform provides services that allow a user to deploy, run and manage web applications. Supported languages include Ruby, Node.js, Java, Python, Clojure and Scala.[[26]](#footnote-27)

## Heroku Applications

With reference to the languages listed in Section 7.1, in Heroku, an application is:

*a collection of source code written in one of these languages, perhaps a framework, and some dependency description that instructs a build system as to which additional dependencies are needed in order to build and run the application.*

In Python, application dependencies are specified by the inclusion of a text file, *requirements.txt,* which lists the dependencies required in order for the application to execute. In general, the source code along with the specified dependencies are all that is required to be deployed to Heroku in order to successfully build and run an application.

### Web Servers

Rather than providing a web server for a web application to run, a Heroku application is required to provide its own web server. There are many web servers available for Python applications, and suitable web servers include Gunicorn (Unix-based systems only), Gevent and Eventlet. [[27]](#footnote-28) These servers are embedded in the application when deployed.

### Deploying to Heroku with Git

Applications are deployed to Heroku using Git, the popular version control system widely used by software developers. Git provides developers with a remote repository for code and other documentation, and provides version control by providing ‘snapshots’ of a project every time a new or updated file is pushed to the remote repository. A remote repository is securely associated with a local repository on a contributor’s local machine.

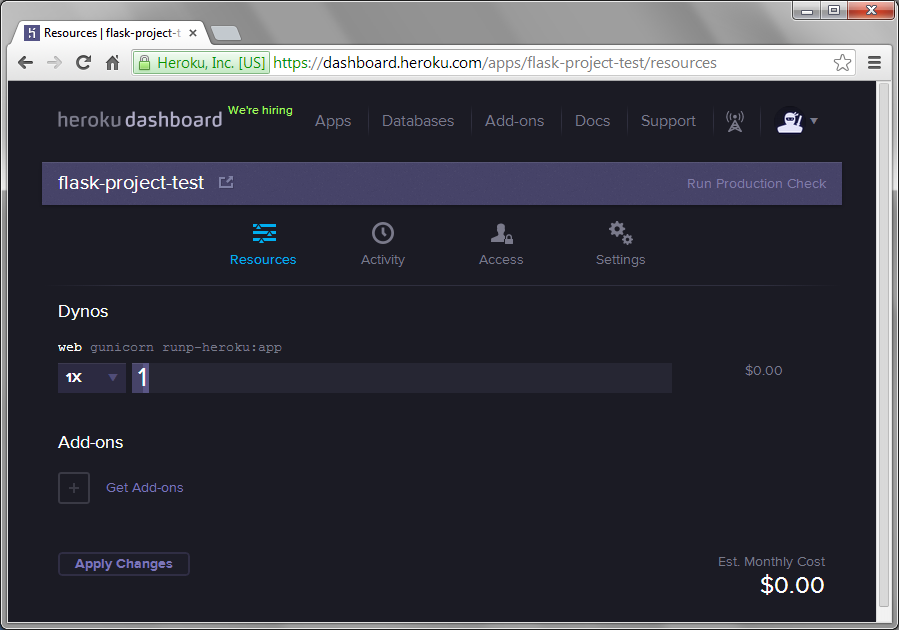
When an application is created on Heroku, a new Git remote repository is associated with the local repository of the application. When the application is pushed to the remote repository for the first time, Heroku commences the build process for the application. The build process involves firstly installing all dependencies listed in the application requirements, and creating any other assets required by the application. Once these are installed, the application is executed.

### Managing Heroku Applications

Heroku executes applications on *dynos*, which are defined as “isolated, secure, virtualised Unix containers that provide the environment to run the application”.[[28]](#footnote-29)

Dynos can easily be scaled in Heroku. A newly created application will automatically be operating on one dyno. As web traffic increases or more processing power required, the dyno count can be scaled upwards accordingly. Heroku provides 750 free dyno-hours per month per application, thus allowing a single-dyno application to run continuously for free. Additional usage incurs charges.

Dyno scaling and other application management functions can be carried out at the command-line, or alternatively by using the GUI Heroku dashboard tool.



**Figure 7.xx – Heroku dashboard showing application running at 1x dyno**

By default, Heroku assigns a random name to projects. The application is hosted at http://<project\_name>.herokuapp.com. The project name can be changed by the user and user URL’s can be assigned to applications as required.

## Use of Heroku with Project Implementation

Heroku will be used in the implementation part of this project. It is intended to create a RESTful web service, implemented in Python using the Flask web framework, and this will be deployed to Heroku in the final implementation. A locally hosted web application will connect with the web service as required.

# Python/Flask Web Application

## Introduction

This section describes the analysis, design, implementation and testing of a web application developed in the Python programming language using the Flask “microframework” for web development.

The implementation was designed such that it would make use of the technologies studied as part of this project.

The web application to be developed is a currency conversion application. The application will allow a client-side user to securely log in through a browser to a web application, hosted locally. The web application will present a selection of currencies to the user, and allow user-input of a monetary amount, with the user selecting the base currency and the converted currency. The user submits the information, and a currency conversion is performed. The converted amount is displayed to the user.

The application will connect to a remotely hosted RESTful web service, posting currency and amount data. The web service will carry out the necessary business logic and return the resulting data to the web application for presentation to the user.

The design for this project has been kept deliberately simple in order to create a suitable working product commensurate with the new technologies explored earlier. It is considered that the proposed implementation will demonstrate a significant knowledge of Python application programming, Python/Flask web application development and remote API/web service deployment to a suitable host in the cloud. The focus is not on what the system does, but rather *how* it does it.

## Technologies

The web application will employ the following technologies in development:

* client-side web application: to be developed in Python employing the Flask web framework
* server-side web service: to be developed in Python employing the Flask web framework. The web service is to comply with the requirements of REST (REpresentational State Transfer) architectural principles
* remote hosting: the web service will be hosted on the Heroku Cloud Application Platform. Heroku provides support for Python applications developed with the Flask web framework
* Git/GitHub: a GitHub remote repository will be used for the duration of the project. Source code and documentation will be pushed to the repository such that a full history of incremental development will be recorded. Heroku also requires the use of Git and GitHub for the deployment of applications
* Notepad++ has been chosen as a simple and lightweight IDE for the development of Python source code for this project
* The application will be developed on a Windows 7 platform, with Python scripts executed using the Python interpreter from Windows Powershell

## Product Delivery

The web application will be developed incrementally. It is considered that this is an appropriate method of delivering the final product, given the concurrent research into the various technologies that is taking place.

The first implementation will be a very simple version of the application, locally hosted, performing minimal calculation. It is intended that increments will be delivered approximately weekly, with each increment adding further functionality/usability/style until a satisfactory solution is implemented which achieves all of the requirements detailed in the following section.

## Analysis

### Requirements Hierarchy Chart

### User Requirements

The requirements of the web application are identified as follows:

* the web app will allow a new user to create a user account for logging into the application
* the web app will allow a previously registered user to securely log in to the application
* the web app will carry out a currency conversion on a specified amount between currencies of the user’s choice

### System Requirements

There will be two modules in the proposed system as follows:

* the client-side web application
* remotely-hosted web service

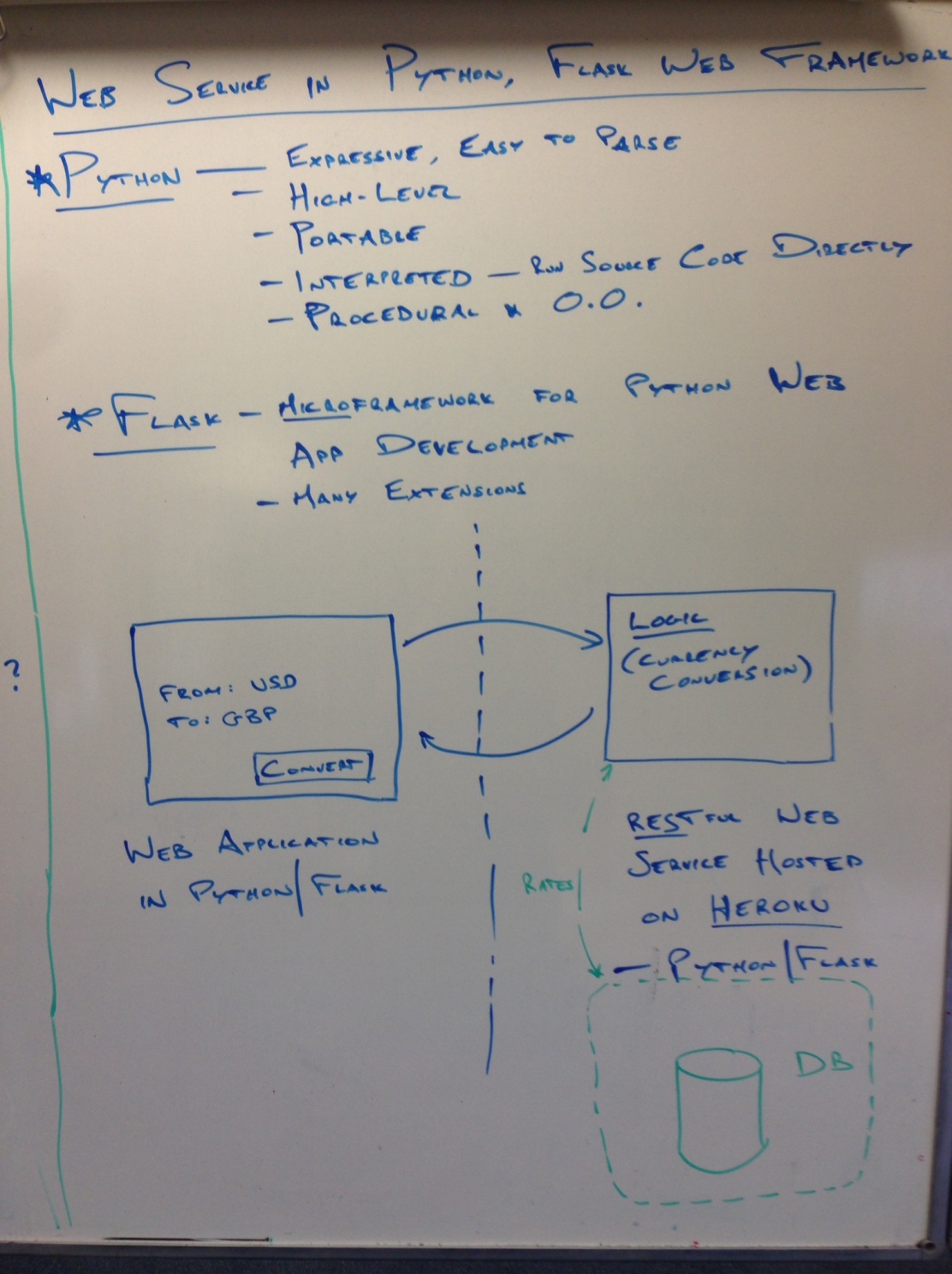
<<System requirement format to be agreed with placement>>

## Design

### Initial Design Development

The initial design of the system was developed to include minimum functionality, and demonstrates the basic web application/web service architecture to be employed in the system. Functionality is initially limited to user input of currencies, with the web service performing the necessary calculation and returning data as required.

Figure 7.xx following is a snapshot of the original design developed in the author’s work placement enterprise.



**Figure 8.xx – initial design**

The design shows the separation of the client-side web application and the remotely hosted web service. The design is explained as follows:

* A web application, developed in Python using the Flask web framework, will be accessed by a user through a browser. The application allows the user to select a base currency from a list, select a target currency from a list and enter a monetary amount to be converted. The text fields presented to the user must only accept a valid monetary amount (i.e. cannot be negative, ust be numeric)
* The inputted data are validated, and posted RESTfully over HTTP to the remotely hosted web service.
* The RESTful web service will be hosted on the Heroku Cloud Application Platform. The service will accept posted data from the web application and perform the required calculation to perform the currency conversion. Currency exchange rates for use in the calculation will be hard-coded into the web service. The converted amount will be posted RESTfully over HTTP to the web application.
* The web application will display the converted amount to the user.
* A database could be used to store currency exchange rates, and may be explored in later increments, depending on implementation progress.

It was also decided to include a user login element to the client-side web application, as this will demonstrate further interesting usage of the capabilities of the Flask web framework.

### Design Summary

The following diagram describes the final design to be implemented in Python using the Flask framework:

RESTful data transfer

**Client-side Web App**

From currency: \_\_\_\_\_\_

To currency: \_\_\_\_\_\_

Amount: \_\_\_\_\_\_

**RESTful Web Service**

*/getcurrencies*

provide list of available currencies

*/convert*

currency conversion

Websdfdeds

RESTful data transfer

CONVERT

* Developed in Python/Flask
* Hosted on Heroku Cloud Application Platform
* Supply and transmit data in JSON format to any interested client
* HTTP basic authorisation required to access exposed API services
* Developed in Python/Flask
* Hosted locally
* Supply data in JSON format to web service (HTTP methods GET / POST)
* User login required

### 

### Implementation

The application will be developed incrementally. The outline plan for incremental development is as follows:

* Initially develop a basic self-contained web application in Python/Flask that possesses minimal functionality and user input and performs some basic calculation. The application should maintain separation between presentation (HTML using Jinja2 template engine) and logic. The application will be self-contained in that the web service will be intrinsically part of the web application. Little or no styling may be included.
* Further iterations will aim to add the following functionalities to the application, in no particular order:
* enhanced user input (selection of currencies from drop-down lists)
* inclusion of user account control and login/logout functions
* separation of web service/API element from client-side application, and deployment of this to the Heroku Cloud Application Platform
* enhanced styling using cascading style sheets
* any other enhancements which may extend the learning opportunities on this project, such as adding database connectivity

Version control is to be used during the development process. Git and GitHub will be used for maintaining version control. Incremental code and documentation development will be regularly ‘pushed’ to a remote repository on GitHub, allowing full historical version control for the application development. The GitHub repository history will demonstrate a continuous work output for the duration of the project and application development.

## Testing Strategy

The testing plan for the project will be developed incrementally as functionality is added to the application. However, as an overall testing strategy the following is to be adopted:

* Unit testing – client-side and server-side modules are to be fully tested during development. Unit testing and results to be appropriately documented.
* Black-box testing – all user input fields are to be fully tested and results compared with expected behavior as follows:
* Numerical fields should be tested for in-range values, boundary values, just out-of-bounds values, out-of-range values, negative values, alphabetical values, non-alpha-numeric values, etc.
* String fields should be tested for empty strings, maximum string length, etc.
* Email fields should be tested for correct email format

In all cases on this project, the agile approach of ‘test-early, test-often’ is assumed.

## Implementation Increments

### Increment #01

The first increment of the application was pushed to GitHub on 07th February 2014. The application runs as a web application hosted on localhost:5000 (or 127.0.0.1:5000), and can be accessed locally by browsing to this address in a web browser.

#### Increment #01 Functionality

The home-page of the application requests the user to input a monetary amount. A nominal currency conversion is carried out and the result displayed to the user. If the user enters an invalid (non-numeric) value, an error message is displayed and the calculation is not carried out. The user can click a button to clear the current conversion if they so wish.

#### Increment #01 Code

The following is the main Python script developed for Increment #01.

PyConvert\_V0.1.py

1 # ICT Project 2014

2 # D Murphy

3 #

4 # PyConvert\_V0.1

5

6 # imports

7 **from** flask **import** Flask**,** request**,** redirect**,** url\_for**,** render\_template

8

9 # configuration

10 DEBUG **=** **True**

11 SECRET\_KEY **=** 'development key'

12

13 # hard-coded currency conversion rates

14 GBP\_TO\_EURO **=** 1.21

15

16 # create the application

17 app **=** Flask**(**\_\_name\_\_**)**

18 app**.**config**.**from\_object**(**\_\_name\_\_**)**

19

20 @app.route**(**'/'**)**

21 **def** render\_home**():**

22 **return** render\_template**(**'base.html'**,** rate**=**GBP\_TO\_EURO**)**

23

24 @app.route**(**'/calculate'**,** methods**=[**'POST'**])**

25 **def** calculate\_values**():**

26 **try:**

27 amount\_as\_string **=** request**.**form**[**'currency'**]**

28 amount **=** float**(**amount\_as\_string**)**

29 converted\_amount **=** GBP\_TO\_EURO **\*** amount

30 **return** render\_template**(**'conversion.html'**,** input**=**amount**,** output**=**converted\_amount**,** rate**=**GBP\_TO\_EURO**)**

31 **except** ValueError**:**

32 error **=** 'You must enter a numerical value! Try again.'

33 **return** render\_template**(**'conversion.html'**,** error**=**error**,** rate**=**GBP\_TO\_EURO**)**

34

35 @app.route**(**'/clear'**,** methods**=[**'POST'**])**

36 **def** clear\_conversion**():**

37 **return** redirect**(**url\_for**(**'render\_home'**))**

38

39 **if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

40 app**.**run**()**

This script represents a simple, self-contained web application. The import statement on line 7 imports some functions from the Flask framework, including the Flask object, of which the application (app) is an instance. The code is self-documenting and so a detailed explanation of each line is not presented. However, some interesting items of note are considered as follows:

* configuration data is included on lines 10/11. The SECRET\_KEY is used for secure sessions, and will be discussed later. DEBUG presents a stack-trace to assist in debugging when an error occurs
* the web application is an instance of the Flask object (line 17)
* ‘decorators’ are used in the script to associate web URL’s with Python functions. This is achieved using the @app.route decorator. When a user browses to the URL in the parameter list of the decorator, the function defined immediately underneath is executed.
* the @app.route decorator is associated with HTTP ‘GET’ methods by default. If the associated web template is to be used for user input, the ‘POST’ method must be included in the parameter list.
* the render\_template function renders a HTML template (Flask uses the Jinja2 template engine). Additional parameters are included as required, depending on the variables used in the template itself.
* the concept of template inheritance was explored in this increment. The base template (base.html) is routed to the URL ‘/’. The template conversion.html extends the base template. These templates are presented below.
* the actual currency conversion is carried out in the calculate\_values function, using a hard-coded exchange rate, provided on line 14. A try-except is used to ensure that valid numerical data is entered by the user, otherwise an error message is returned to the template.

The following are the HTML templates (base.html and conversion.html) for this increment:

base.html

**1** <!-- base.html

**2**  ICT Project

**3** D Murphy -->

**4**

**5** <!doctype html>

**6** <head>

**7** <title>**PyConvert**</title>

**8** </head>

**9** <body>

**10** <div class=page>

**11** <h1>**PyConvert(V0.1)**</h1>

**12** <p>**Enter amount in GBP in form below. PyConvert will return Euro value (1GBP = {{ rate }} Euro)**</p>

**13**

**14** <form action=**"/calculate"** method=post>

**15** <dl>

**16** <dt>**Amount in GBP:**

**17** <dd><input type=text name=currency>

**18** <dd><input type=submit value=**"Calculate"**>

**19** </dl>

**20** </form>

**21**

**22 {% block calc %}{% endblock %}**

**23**

**24** </div>

**25** </body>

conversion.html

**1** <!-- conversion.html

**2**  ICT Project

**3** D Murphy -->

**4**

**5 {% extends "base.html" %}**

**6 {% block calc %}**

**7 {% if output %}**

**8** <p><strong>**{{ input }}(GPB) in Euro is: {{ output }}** </strong></p>

**9 {% elif error %}**

**10** <p><strong>**{{ error }}**</strong></p>

**11 {% endif %}**

**12**

**13** <hr>

**14** <form action=**"/clear"** method=post>

**15** <input type=submit value=**"Clear conversion"**>

**16** </form>

**17 {% endblock %}**

The templates demonstrate template inheritance (conversion.html extends base.html) and the use of Python-like code, such as **if-elif-endif**, within the templates (provided for by the Jinja2 template engine).

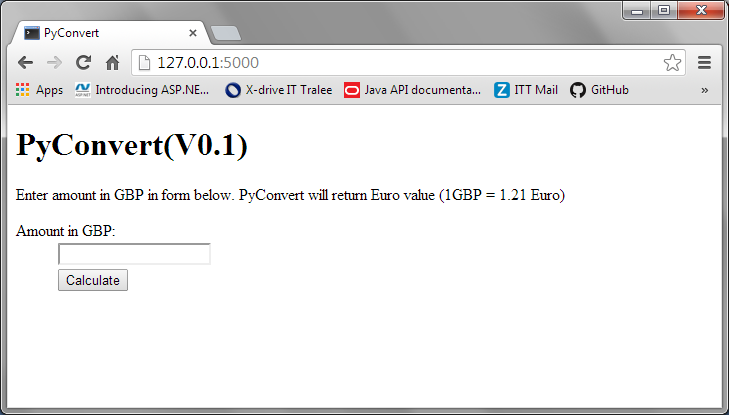
#### Increment #01 Testing

The following user input values are tested and compared with expected behavior:

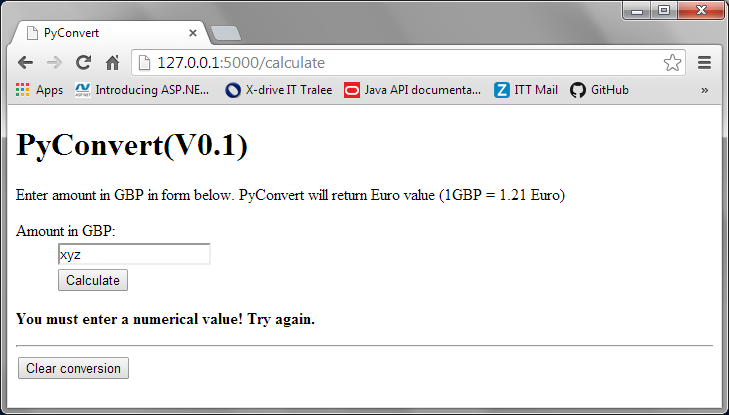
|  |  |  |  |
| --- | --- | --- | --- |
| Test Ref | User Input | Test Value | Expected Result |
| 1.01 | Conversion amount | Non-numerical value (‘xyz’) | Error message |
| 1.02 | Conversion amount | Empty string | Error message |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

#### Increment #01 Output

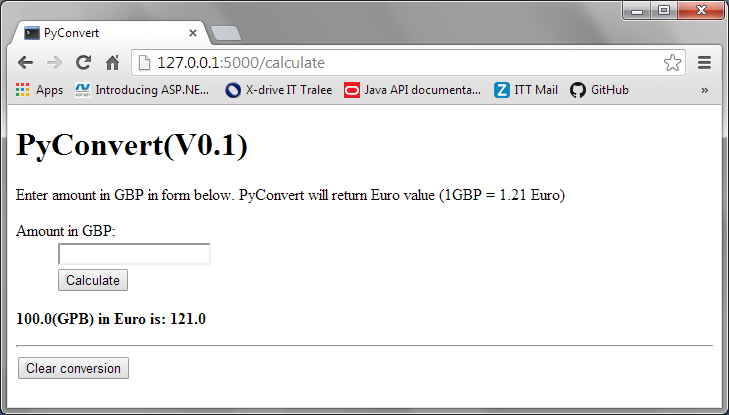
The following figures show screen output from a typical session using the web application.



**Figure 8.xx – Increment #01 web application**



**Figure 8.xx – Increment #01 with non-valid test data**



**Figure 8.xx – Increment #01 with valid test data**

### Increment #02

The second main increment of the application was pushed to GitHub on 13th February 2014. Again the application runs as a web application hosted on localhost:5000 (or 127.0.0.1:5000), and can be accessed locally by browsing to this address in a web browser.

#### Increment #02 Functionality

The code from the first increment was re-written in order to move towards separating presentation from the business logic in an effort to move towards the *Model-View-Controller* software pattern.

The project directory was reorganised to mirror that recommended in the Flask documentation [[29]](#footnote-30)(which is that of a Python module), and this structure is as follows:

/WebApp

/runserver.py

/\_documentation/

/[various project documentation]

/\_source\_code

/[python .py scripts]

/static/

/css/

[.css files]

/templates

[html templates]

When the application is started, browsing to the home URL ‘/’ presents a home page for the application, with a link and a submit button to direct the user to the conversion application. It is noted on the home page that this may be a login page in a future increment.

The currency conversion application allows the user to:

* select a base currency from a dropdown list of pre-loaded currencies
* select a currency for the currency conversion
* input a monetary amount to convert

If the user submits a valid numerical amount, the conversion is displayed on a new page. The user can click a link to carry out another conversion, or alternatively to return to the home page.

If invalid input is entered, helpful messages are ‘flashed’ on the webpage to direct the user accordingly.

The application has also benefitted from css styling.

#### Increment #02 Code

New Python scripts routes.py, forms.py and rates.py were created. Routes.py contains the URL mapping to functions which control data passing to the templates.

Forms.py contains form objects for form input using the Flask-WTF extension. It is useful to be able to define forms in this manner as the form can be defined in the Python code, with Flask-WTF generating the associated HTML. In addition, Flask-WTF provides useful methods for form validation.[[30]](#footnote-31)

Rates.py separates the business logic from the presentation layer. A dictionary of hard-coded exchange rates (relative to Euro = 1.0) is provided for currency conversion. Various functions are defined, including functions for returning an exchange rate between two given currencies and for calculating the currency conversion and returning the result. The separation of this part of the code from the presentation layer represents the first move towards the design goal of implementing the business logic as a RESTful API. For now, all the code required in order for the application to function is self-contained.

Two further Python scripts are included; \_\_init\_\_.py, which creates the Flask instance and contains application configuration data (in accordance with the Python module structure discussed earlier), and runserver.py, located in the project root directory, which kickstarts the application.

runserver.py

1 **from** \_source\_code **import** app

2

3 app**.**run**(**debug**=True)**

\_\_init\_\_.py

1 **from** flask **import** Flask

2

3 # config

4 app **=** Flask**(**\_\_name\_\_**)**

5 app**.**secret\_key **=** 'development key'

6

7 # import routes

8 **import** routes

routes.py

1 # ICT Project 2014

2 # D Murphy

3 #

4 # routes.py

5

6 # imports

7 **from** \_source\_code **import** app

8 **from** flask **import** Flask**,** request**,** redirect**,** url\_for**,** render\_template**,** flash

9 **from** forms **import** ConversionForm**,** ResultForm**,** EnterForm

10 **from** rates **import** getRate**,** getSpecificRate**,** exchange

11

12 @app.route**(**'/'**,** methods**=[**'GET'**,** 'POST'**])**

13 **def** home**():**

14 form **=** EnterForm**()**

15

16 **if** request**.**method **==** 'POST'**:**

17 **return** redirect**(**url\_for**(**'conversion'**))**

18

19 **elif** request**.**method **==** 'GET'**:**

20 **return** render\_template**(**'home.html'**,** form**=**form**)**

21

22 @app.route**(**'/convert'**,** methods**=[**'GET'**,** 'POST'**])**

23 **def** conversion**():**

24 form **=** ConversionForm**()**

25

26 **if** request**.**method **==** 'POST'**:**

27 **if** form**.**validate**()** **==** **False:**

28 form**.**conversionAmount**.**data **=** ''

29 **return** render\_template**(**'conversion.html'**,** form**=**form**)**

30 **else:**

31 unformatted\_input\_amount **=** float**(**form**.**conversionAmount**.**data**)**

32 from\_currency **=** form**.**fromCurrency**.**data

33 to\_currency **=** form**.**toCurrency**.**data

34

35 converted\_amount **=** exchange**(**from\_currency**,** to\_currency**,** unformatted\_input\_amount**)**

36 specific\_rate **=** getSpecificRate**(**from\_currency**,** to\_currency**)**

37

38 input\_amount **=** '%.2f' **%** **(**unformatted\_input\_amount**)**

39

40 **return** render\_template**(**'conversion.html'**,** form**=**form**,** input\_amount**=**input\_amount**,**

41 from\_currency**=**from\_currency**,** to\_currency**=**to\_currency**,**

42 converted\_amount**=**converted\_amount**,** specific\_rate**=**specific\_rate**)**

43

44 **elif** request**.**method **==** 'GET'**:**

45 **return** render\_template**(**'conversion.html'**,** form**=**form**)**

forms.py

1 **from** flask**.**ext**.**wtf **import** Form

2 **from** wtforms **import** TextField**,** SubmitField**,** SelectField**,** validators**,** ValidationError

3 **from** rates **import** populateCurrencyList

4

5

6 **class** **ConversionForm(**Form**):**

7 fromCurrency **=** SelectField**(**'From currency:'**,** choices**=**populateCurrencyList**())**

8 toCurrency **=** SelectField**(**'To currency:'**,** choices**=**populateCurrencyList**())**

9

10 conversionAmount **=** TextField**(**"Enter amount to convert:"**,** **[**validators**.**Required**(**"Please enter a conversion amount"**)])**

11 convertButton **=** SubmitField**(**"Convert"**)**

12

13 **def** \_\_init\_\_**(**self**,** **\***args**,** **\*\***kwargs**):**

14 Form**.**\_\_init\_\_**(**self**,** **\***args**,** **\*\***kwargs**)**

15

16 **def** validate**(**self**):**

17 **if** **not** Form**.**validate**(**self**):**

18 **return** **False**

19

20 **try:**

21 amount **=** float**(**self**.**conversionAmount**.**data**)**

22 **return** **True**

23 **except** ValueError**:**

24 self**.**conversionAmount**.**errors**.**append**(**'You must enter a numerical value! Try again.'**)**

25 **return** **False**

26

27 **class** **ResultForm(**Form**):**

28 clearButton **=** SubmitField**(**"Another conversion"**)**

29

30 **class** **EnterForm(**Form**):**

rates.py

1 # ICT Project 2014

2 # D Murphy

3 #

4 # rates.py

5 # exchange rates and conversion functions

6 # getSpecificRate() and exchange() return Strings

7 # getRate() returns a float

8

9 # populateCurrencyList() returns a list of currencies from

10 # dicts in exchange\_rates to populate WTForms SelectFields

11

12

13 # exchange rates are relative to 1.0 \* Euro

14 exchange\_rates **=** **{**'EUR'**:** 1.0**,**

15 'USD'**:** 1.36379**,**

16 'GBP'**:** 0.82910**,**

17 'AUD'**:** 1.51202**,**

18 'CAD'**:** 1.50140**,**

19 'NZD'**:** 1.64072**}**

20

21 **def** populateCurrencyList**():**

22 currencyList **=** **[]**

23 **for** currency **in** exchange\_rates**:**

24 currencyList**.**append**((**currency**,** currency**))**

25

26 **return** currencyList

27

28 **def** getRate**(**currency**):**

29 **if** currency **in** exchange\_rates**:**

30 **return** exchange\_rates**[**currency**]**

31

32 **def** getSpecificRate**(**from\_currency**,** to\_currency**):**

33 from\_rate **=** exchange\_rates**[**from\_currency**]**

34 to\_rate **=** exchange\_rates**[**to\_currency**]**

35

36 **return** '%.5f' **%** **(**to\_rate **/** from\_rate**)**

37

38 **def** exchange**(**from\_currency**,** to\_currency**,** amount**):**

39 from\_rate **=** exchange\_rates**[**from\_currency**]**

40 to\_rate **=** exchange\_rates**[**to\_currency**]**

41

42 **return** '%.2f' **%** **(**amount **\*** to\_rate **/** from\_rate**)**

The HTML templates used in the project were also rewritten, and now include of a ‘baseLayout.html’, which serves as a foundation layout for all other web templates (home.html and conversion.html).

baseLayout.html

**1** <!-- baseLayout.html

**2**  ICT Project

**3** D Murphy -->

**4**

**5** <!doctype html>

**6** <html>

**7** <head>

**8** <title>**PyConvert**</title>

**9** <link rel=**"stylesheet"** href=**"{{ url\_for('static', filename='css/webapp.css') }}"**>

**10** </head>

**11** <body>

**12** <header>

**13** <div class=**"container"**>

**14** <h1 class=**"logo"**>**PyConvert(V0.1)**</h1>

**15** <nav>

**16** <ul class=**"menu"**>

**17** <li><a href=**"{{ url\_for('home') }}"**> **Home**</a></li>

**18** <li><a href=**"{{ url\_for('conversion') }}"**> **Convert!**</a></li>

**19** </ul>

**20** </nav>

**21** </div>

**22** </header>

**23**

**24** <div class=**"container"**>

**25 {% block content %}**

**26 {% endblock %}**

**27** </div>

**28** </body>

home.html

**1 {% extends "baseLayout.html" %}**

**2**

**3 {% block content %}**

**4** <div class=**"jumbo"**>

**5** <h2>**Welcome to PyConvert Web App**</h2>

**6** <h3 class=**"home"**>**[[This might be a future login page for secure login]]**</h3>

**7** <hr>

**8** </div>

**9**

**10** <form action=**"{{ url\_for('home') }}"** method=**"post"**>

**11 {{ form.hidden\_tag() }}**

**12**

**13 {{ form.enterButton() }}**

**14** </form>

**15 {% endblock %}**

conversion.html

**1** <!-- conversion.html

**2**  ICT Project

**3**  D Murphy -->

**4**

**5 {% extends "baseLayout.html" %}**

**6**

**7 {% block content %}**

**8 {% if converted\_amount %}**

**9** <h3><strong>**{{ input\_amount }} {{ from\_currency}} = {{ converted\_amount }} {{ to\_currency }}** </strong></h3>

**10** <p id=**"conversion"**>**[1.00 {{ from\_currency }} = {{ specific\_rate }} {{ to\_currency }}]**</p>

**11** <br/>

**12** <div class=**"convertPanel"**>

**13** <a href=**"{{ url\_for('conversion') }}"** class=**"convertLink"**>**Another Conversion**</a>

**14** </div>

**15 {% else %}**

**16**

**17** <h3>**Select from currencies and enter conversion amount**</h3>

**18** <hr>

**19**

**20 {% for message in form.conversionAmount.errors %}**

**21** <div class=**"flash"**>**{{ message }}**</div>

**22 {% endfor %}**

**23**

**24** <form action=**"{{ url\_for('conversion') }}"** method=**"post"**>

**25 {{ form.hidden\_tag() }}**

**26**

**27 {{ form.fromCurrency.label }}**

**28 {{ form.fromCurrency }}**

**29**

**30 {{ form.toCurrency.label }}**

**31 {{ form.toCurrency }}**

**32**

**33 {{ form.conversionAmount.label }}**

**34 {{ form.conversionAmount(autocomplete="off") }}**

**35**

**36 {{ form.convertButton }}**

**37 </form>**

**38**

**39 {% endif %}**

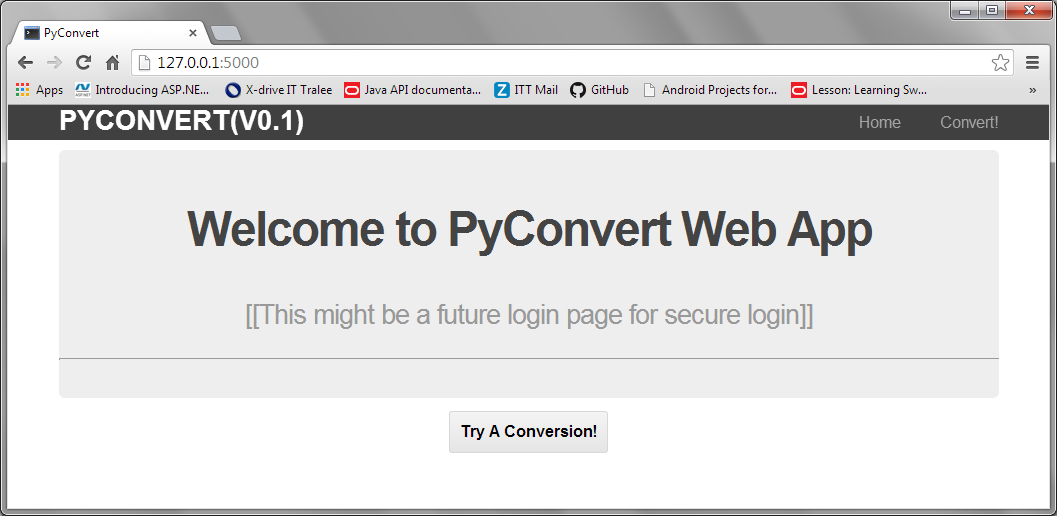
#### Increment #02 Testing

The following user input values are tested and compared with expected behavior:

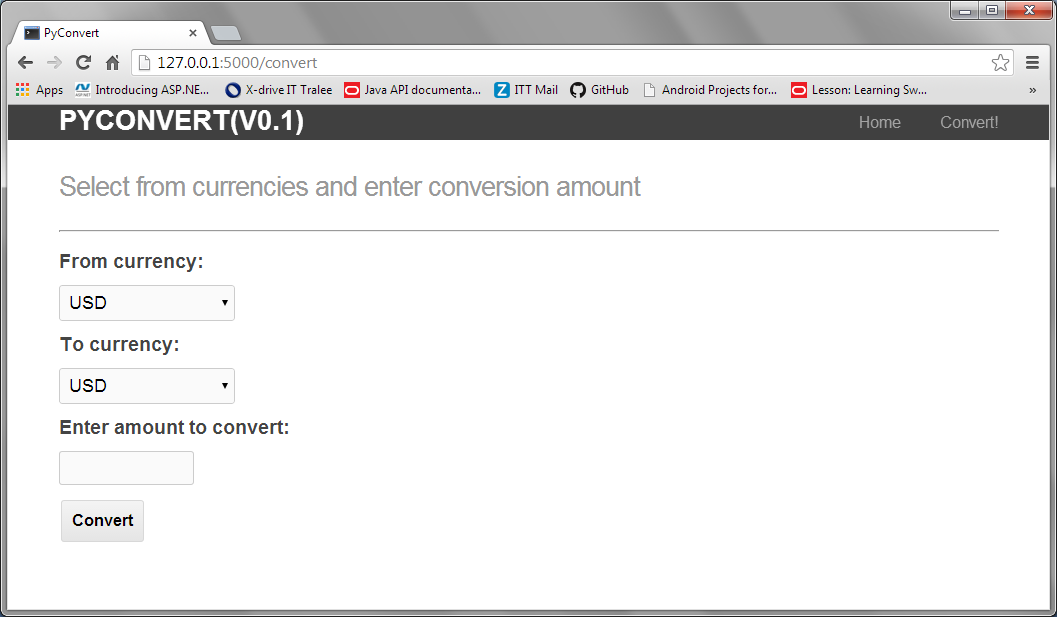
|  |  |  |  |
| --- | --- | --- | --- |
| Test Ref | User Input | Test Value | Expected Result |
| 2.01 | Conversion amount | Non-numerical value (‘xyz’) | Error message |
| 2.02 | Conversion amount | Empty string | Error message |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

#### Increment #02 Output

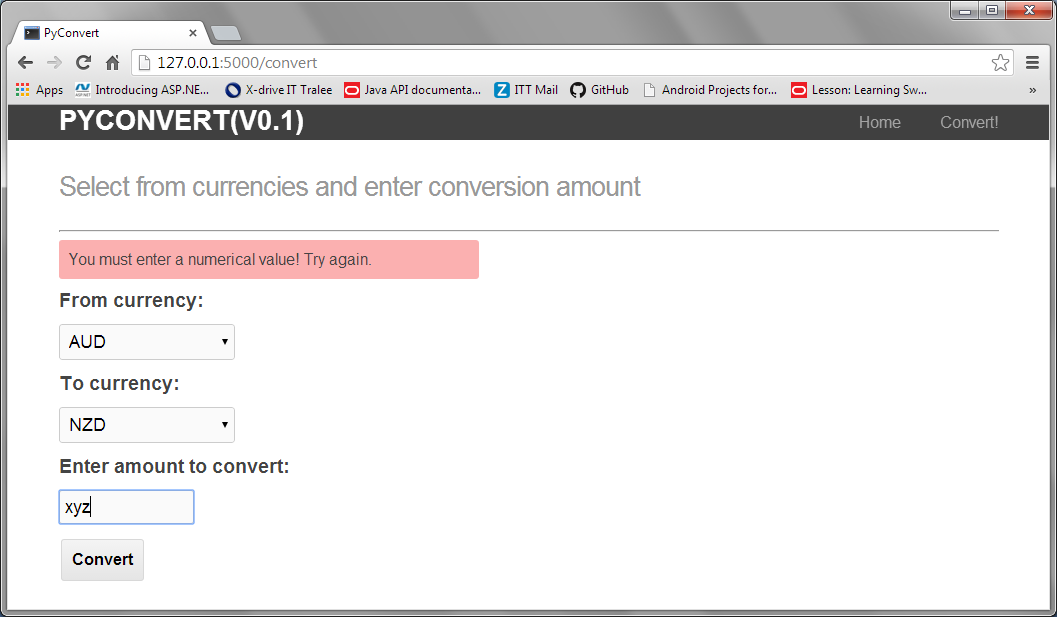
The following figures show screen output from a typical session using the web application.



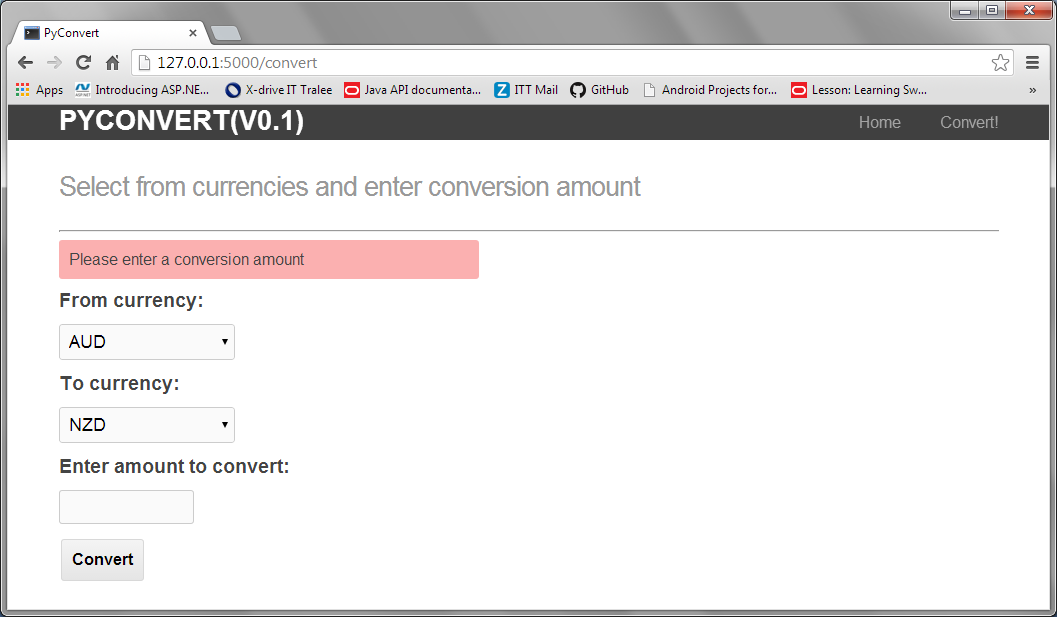
**Figure 8.xx – Increment #02 home page**

****

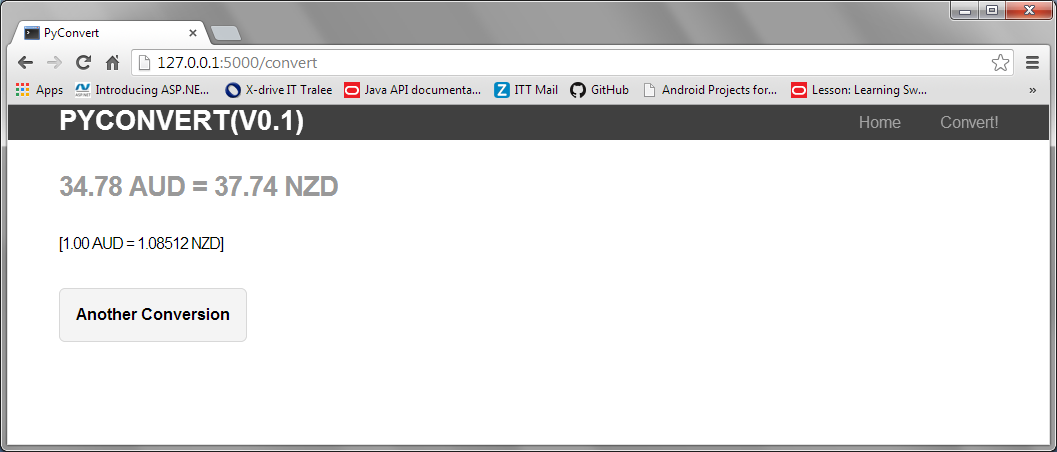
**Figure 8.xx – Increment #02 conversion application**

****

**Figure 8.xx – Increment #02 with invalid test data**

****

**Figure 8.xx – Increment #02 with invalid test data**



**Figure 8.xx – Increment #02 currency conversion output**

### Increment #03

The goal of the third increment in the development of the application was to successfully deploy a sample python application to the Heroku cloud application platform. The ultimate goal of the implementation is to host a RESTful web service on Heroku and as such it is considered appropriate to try to get a sample application running on Heroku prior to developing the RESTful web service.

#### Sample Deployment to Heroku

Instructions are provided on the Heroku website for deploying, building and executing a web application developed in Python using the Flask web framework (instructions can be found at <https://devcenter.heroku.com/articles/getting-started-with-python>).

A user must authenticate their machine using SSH on first login to Heroku. A Python project must be created inside a “virtual environment”, using Virtualenv. Virtualenv is a tool used “to create isolated environments for Python in which you can install packages without interfering with the other virtualenvs nor with the system Python's packages”.[[31]](#footnote-32) All dependencies (i.e. Python and Flask extensions) are installed within the virtual environment and are specific to the Python project in question. Gunicorn was chosen as the web server. The sample application was developed in a Linux environment (Ubuntu 12.04) as Gunicorn is only available for Unix systems.

A simple ‘Hello World’-type Python-Flask web application was created in the project folder. Dependencies were specified in requirements.txt, and the Gunicorn web server and point of entry to the application were specified in a special text file called Procfile. The application was then deployed to Heroku via Git and successfully executed.

#### Deployment of Implementation to Heroku

As another exercise in deploying to Heroku it was decided to attempt to deploy the complete second increment (Increment #02 as discussed previously) as a single application to Heroku. Following the above procedures, the application was successfully hosted at <http://flask-project-test.herokuapp.com>.

In order to execute the application on the Gunicorn web server a slight change was required to runserver.py, which is now renamed as runp-heroku.py as follows:

1 **from** \_source\_code **import** app

Procfile specifies the point of entry and Gunicorn web server, and is as follows:

1 web: gunicorn runp-heroku:app

Finally, requirements.txt specifies the dependencies to be installed while building the application on Heroku:

1 Flask==0.10.1

2 Flask-WTF==0.9.4

3 Jinja2==2.7.2

4 MarkupSafe==0.18

5 WTForms==1.0.5

6 Werkzeug==0.9.4

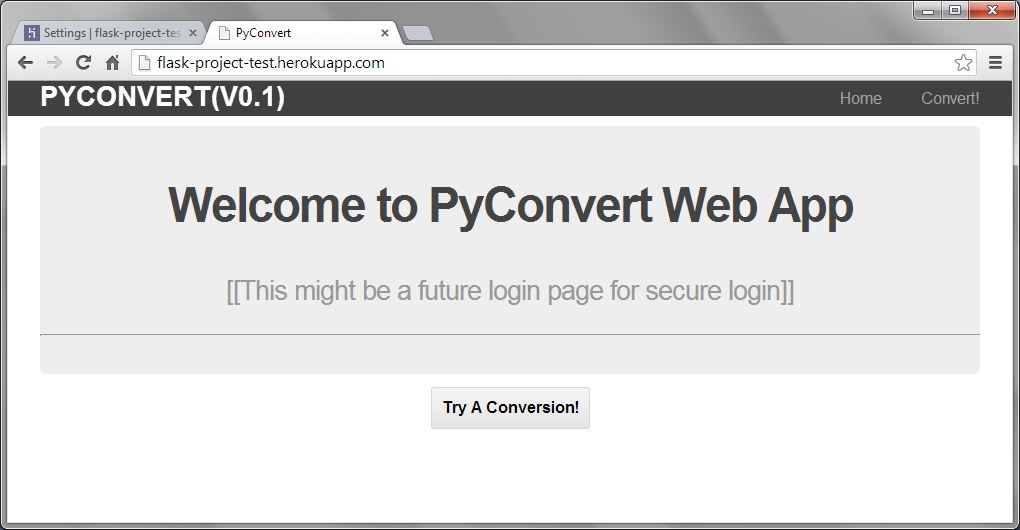
7 argparse==1.2.1

8 gunicorn==18.0

9 itsdangerous==0.23

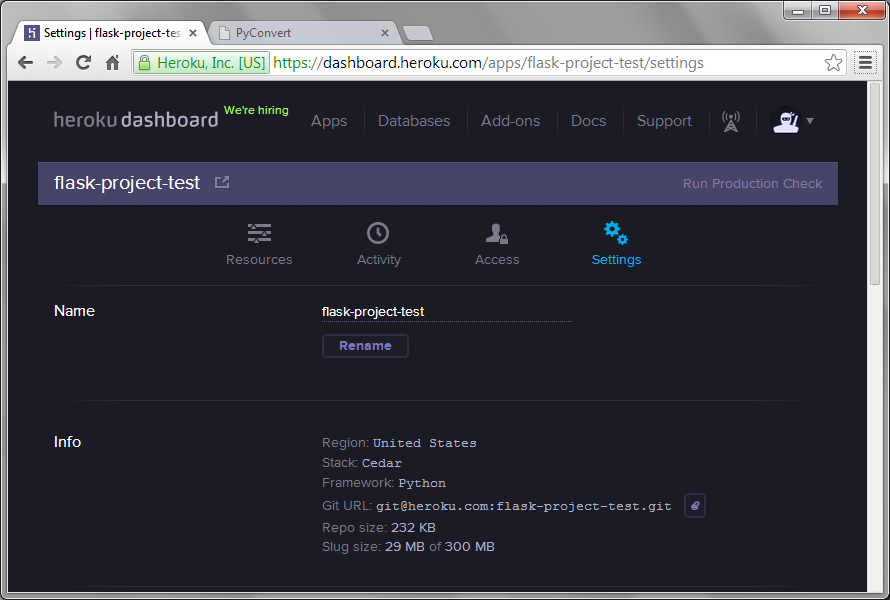
10 wsgiref==0.1.2

A sample execution of the application running on the Heroku platform is shown in Figure 8.xx below; note the URL of the application with the herokuapp.com suffix.



**Figure 8.xx – execution of application on Heroku platform**

Figure 8.xx following shows the Heroku dashboard with details of the sample application deployed.



**Figure 8.xx – Heroku dashboard showing details of sample deployment**

#### Increment #03 Testing

Testing of this increment was identical to that for Increment #02 as described in Section 8.7.2.3.

### Increment #04

Increment #04 was pushed to GitHub on 24th February 2014. The aim of this increment was to create a RESTful web service (or API) that would interact with a web application over HTTP. It was decided to encode all data in JSON format.

The API created in this increment was tested using cURL (on a Linux/Ubuntu platform), which is a command-line tool for transferring data using various protocols, and in this case HTTP methods. cURL allows the user to construct HTTP methods (e.g. GET, POST, PUT, DELETE etc) and specify the data format, associated parameters, user authentication data, etc.

#### Increment #04 functionality

The API to be developed is to take the functionality previously provided in rates.py (see Section 8.7.2.2) and enable RESTful transfer of JSON data to and from the API. The API must be capable of being independently hosted, allowing access from any interested authorised client-side application.

#### Increment #04 code

The code in the newly named exchange\_api.py consists of the code formerly in rates.py adjusted to function as a RESTful API, and is as follows:

1 # RESTful api (Flask/Python)

2

3 # currency conversion

4 # all data in HTTP methods is JSON format

5

6 # getSpecificRate() and exchange() return floats

7 # getRate() returns a float

8

9 # get\_active\_currency\_list() returns currency

10 # dictionary (JSON format)

11

12 # exchange rates are relative to 1.0 \* Euro

13

14 # code structure adapted from tutorial at

15 # <http://blog.miguelgrinberg.com/post/designing-a-restful-api-with-> python-and-flask

16 # by Miguel Grinberg

17

18

19 **from** flask **import** Flask**,** jsonify**,** abort**,** request**,** make\_response

20 **from** flask**.**ext**.**httpauth **import** HTTPBasicAuth

21

22

23 app **=** Flask**(**\_\_name\_\_**)**

24 auth **=** HTTPBasicAuth**()**

25

26

27 # hard-coded exchange rates

28 exchange\_rates **=** **{**'EUR'**:** 1.0**,**

29 'USD'**:** 1.36379**,**

30 'GBP'**:** 0.82910**,**

31 'AUD'**:** 1.51202**,**

32 'CAD'**:** 1.50140**,**

33 'NZD'**:** 1.64072**}**

34

35

36 # user authorisation

37 @auth.get\_password

38 **def** get\_password**(**username**):**

39 **if** username **==** 'daithi'**:**

40 **return** 'pass1234'

41 **return** **None**

42

43

44 @auth.error\_handler

45 **def** unauthorized**():**

46 **return** make\_response**(**jsonify**(** **{** 'error'**:** 'Unauthorised access' **}** **),**

403**)**

47

48

49 # url to return specific exchange rate

50 @app.route**(**'/testapi/convert/<string:currency>'**,** methods **=** **[**'GET'**])**

51 #@auth.login\_required

52 **def** get\_rate**(**currency**):**

53 **if** currency **in** exchange\_rates**:**

54 **return** jsonify **(** **{** 'rate'**:** exchange\_rates**[**currency**]** **}** **)**

55 **else:**

56 abort**(**404**)**

57

58

59 # url to return currency conversion based on JSON request parameters

60 @app.route**(**'/testapi/convert'**,** methods **=** **[**'POST'**])**

61 @auth.login\_required

62 **def** convert\_amount**():**

63 **if** **not** request**.**json**:**

64 abort**(**400**)**

65 **if** **not** 'from\_currency' **in** request**.**json**:**

66 abort**(**400**)**

67 **if** **not** 'to\_currency' **in** request**.**json**:**

68 abort**(**400**)**

69 **if** **not** 'amount' **in** request**.**json**:**

70 abort**(**400**)**

71

72 **if** **not** validate\_amount**(**request**.**json**[**'amount'**]):**

73 abort**(**400**)**

74

75 **if** request**.**json**[**'from\_currency'**]** **not** **in** exchange\_rates**:**

76 abort**(**404**)**

77 **if** request**.**json**[**'to\_currency'**]** **not** **in** exchange\_rates**:**

78 abort**(**404**)**

79

80 converted\_amount **=** exchange**(**request**.**json**[**'from\_currency'**],**

81 request**.**json**[**'to\_currency'**],**

82 request**.**json**[**'amount'**])**

83

84 specific\_rate **=** get\_unit\_rate**(**request**.**json**[**'from\_currency'**],**

85 request**.**json**[**'to\_currency'**])**

86

87 **return** jsonify**(** **{** 'converted\_amount'**:** converted\_amount**,**

88 'unit\_rate'**:** specific\_rate **}** **)**

89

90

91 # url to return dictionary of currencies in JSON

92 @app.route**(**'/testapi/getcurrencies'**,** methods **=** **[**'GET'**])**

93 @auth.login\_required

94 **def** get\_active\_currency\_list**():**

95 currency\_list **=** **[]**

96 **for** currency **in** exchange\_rates**:**

97 currency\_list**.**append**((**currency**,** currency**))**

98

99 **return** jsonify **(** **{** 'currency\_list'**:** currency\_list **}** **)**

100

101

102 # JSON error handlers

103 @app.errorhandler**(**404**)**

104 **def** not\_found**(**error**):**

105 **return** make\_response**(**jsonify**(** **{** 'error'**:** 'Resource not found' **}**

**),** 404**)**

106

107

108 @app.errorhandler**(**400**)**

109 **def** bad\_request**(**error**):**

110 **return** make\_response**(**jsonify**(** **{** 'error'**:** 'Bad request' **}** **),** 400**)**

111

112

113 # utility methods

114 **def** get\_unit\_rate**(**from\_currency**,** to\_currency**):**

115 from\_rate **=** exchange\_rates**[**from\_currency**]**

116 to\_rate **=** exchange\_rates**[**to\_currency**]**

117

118 **return** to\_rate **/** from\_rate

119

120

121 **def** exchange**(**from\_currency**,** to\_currency**,** amount**):**

122 from\_rate **=** exchange\_rates**[**from\_currency**]**

123 to\_rate **=** exchange\_rates**[**to\_currency**]**

124

125 **return** amount **\*** to\_rate **/** from\_rate

126

127

128 **def** validate\_amount**(**amount**):**

129 **if** isinstance**(**amount**,** basestring**):**

130 **return** **False**

131 **else:**

132 **try:**

133 float**(**amount**)**

134 **return** **True**

135 **except** ValueError**:**

136 **return** **False**

137

138

139 # execute api on port 8000

140 **if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

141 app**.**run**(**debug **=** **True,** port **=** 8000**)**

The script first defines hard-coded currency exchange rates for use in conversion calculations. URL’s are defined (/testapi/convert and /testapi/getcurrencies) which, when accessed from a web application, return a currency conversion or a list of currencies, both in JSON format. There is also validation taking place in which the incoming request parameters are checked firstly if they are in JSON format, and secondly if they match the expected parameters.

The convert\_amount() function takes a Python dict as input in the following format: {‘from\_currency’: from\_currency, ‘to\_currency’: to\_currency, ‘amount’: amount}, where from\_currency and to\_currency values are strings, and amount value is numerical.

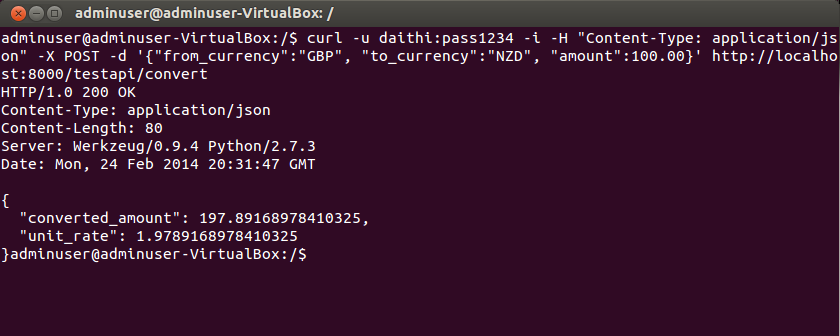
The API also makes use of the HTTPAuth extension to include user authorisation in the HTTP methods. A hard-coded username and password are supplied in the get\_password() function for the purposes of this increment. Any route in the script which requires authorisation is decorated with @auth.login\_required decorator.

Finally, the API is executed on lines 140/141 on localhost, port 8000.

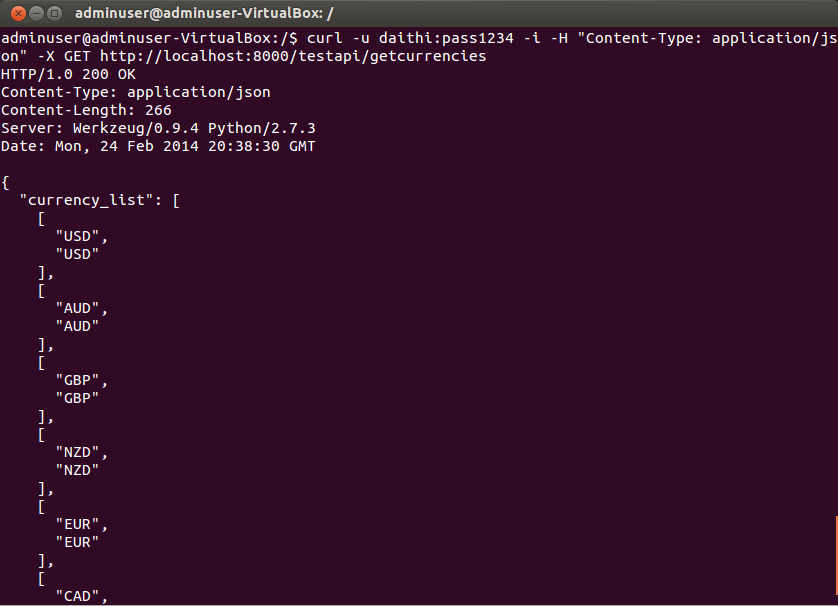
#### Increment #04 testing

Testing of the API routes is carried out using cURL. A full description of cURL commands is beyond the scope of this document, but a typical command will include the HTTP method required, HTTP header, user authorisation data and URL parameters in JSON format. The two URL’s defined in the API are tested for correct and incorrect user authorisation data, and correct and incorrect parameters.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Ref | User Input | URL / HTTP Method | Expected Result |
| 4.01 | Username: daithi  Password: pass1234  from\_currency: ‘GBP’  to\_currency: ‘NZD’  amount: 100.00 | /testapi/convert  POST | 200 OK  Converted amount & unit rate |
| 4.02 | Username: daithi  Password: pass1234  from\_currency: ‘GBP’  to\_currency: NULL  amount: 100.00 | /testapi/convert  POST | 400 Bad request |
| 4.03 | Username: daithi  Password: pass1234  from\_currency: NULL  to\_currency: ‘USD’  amount: 100.00 | /testapi/convert  POST | 400 Bad request |
| 4.04 | Username: daithi  Password: pass1234  from\_currency: ‘CAD’  to\_currency: ‘USD’  amount: ‘100.00’ | /testapi/convert  POST | 400 Bad request |
| 4.05 | Username: daithi  Password: xyz  from\_currency: ‘CAD’  to\_currency: ‘USD’  amount: 100.00 | /testapi/convert  POST | 403 Forbidden |
| 4.06 | Username: xyz  Password: pass1234  from\_currency: ‘CAD’  to\_currency: ‘USD’  amount: ‘100.00’ | /testapi/convert  POST | 403 Forbidden |
| 4.07 | No username or password  from\_currency: ‘CAD’  to\_currency: ‘USD’  amount: ‘100.00’ | /testapi/convert  POST | 403 Forbidden |
| 4.08 | Username: daithi  Password: pass1234 | /testapi/getcurrencies  GET | 200 OK  List of exchange rates |



**Figure 8.xx – execution of cURL test ref 4.01**



**Figure 8.xx – execution of cURL test ref 4.08**

All tests were found to produce the expected test results.

### Increment #05

The main aim of Increment #05 of the implementation was to modify the client-side Flask application to call into the exposed services of the server-side API.

The client-side application was modified and improved to ensure robust interfacing with the API. Additional functionality was developed to allow for displaying currency labels in addition to currency codes.

Functionality of the API was improved to include periodic connection to a live currency feed to download up-to-date currencies in JSON format for use in currency conversions.

#### Increment #05 client-side application modifications

The main change to the application in this increment was to modify the code so that the application would interface RESTfully with the API. This was achieved by creating a new script, currencies.py, which defines a new class Currencies(), which handles interfacing with the live currency feed service exposed by the API. A method get\_currency\_list() retrieves the list of currencies being offered by the API. Another method, test\_feed\_connection() tests the connection to the API, and deals with any HTTP errors appropriately.

In the convert() function of routes.py, an instance of Currencies() is created and the test\_feed\_connection() method is called to check that the required connection is open. Once an authorised connection is established (username and password data hardcoded at present) the ConversionForm object is populated with currencies retrieved from the exposed API /getcurrencies URL.

A HTTP ‘POST’ request is then made (using the Flask ‘Requests’ extension) to the API service that performs the currency conversion. Parameters in JSON format are included in the HTTP request, in the format expected by the API (incorrect or missing parameters will result in an appropriate error response from the API). The HTTP JSON response is decoded and routed to the relevant HTML template for presentation to the user.

Additional HTML templates are also created to which the user is redirected on the occurrence of HTTP errors (400, 403 and 404), such as when a user is not authorised or the API is not exposed.

Finally, any numerical values used in the application are in the Decimal format. The Decimal type provides accuracy in numerical calculations that a binary float type cannot provide, and therefore is much more suitable for use in monetary applications where accuracy is vital.

The client-side application runs on localhost port 5000.

#### Increment #05 API modifications

The first major change to the API in this increment was the separating of the API code into its own Python project, with associated project directory structure. A distinct GitHub repository was created for the API, such that the project now consists of two separate Python applications, the client-side app and the API web service.

Functionality was added to the API to periodically download live currency rates for use in the web service. The live currency feed used is a free service providing rates in JSON format for ten currencies relative to 1.0 EUR (rates may have a 20-minute delay). The API uses this feed to carry out a currency conversion between any of the currencies provided by the feed. The exchange() function in exchange\_api.py carries out a currency conversion by converting first to Euro, and then to the currency of choice. It is accepted that this is not a true conversion, but is considered accurate enough for the purposes of this application.

In order to download live rates periodically, a script (rates\_thread.py) is created and defines a RatesThread() class. The class inherits from the Python Thread class, and contains a run() method which executes when the thread object is started within the main process. The method executes periodically (set to 300 seconds, or 5 minutes at present) and populates a Python dictionary with the currency code, rate relative to 1.0 EUR and currency label. This data is then used by the main process in its exposed services. The time and date of the last live feed update is also recorded.

Similarly to the client-side application, all numerical values used in the code are in Decimal format which is suitable for use in monetary applications.

While the client-side application runs on port 5000, the API runs locally on localhost port 8000, and so the two applications can successfully communicate.

#### Increment #05 code – client-side application

Code is provided here for routes.py and currencies.py which demonstrate the major modifications in the application since the last increment.

routes.py

1 # ICT Project 2014

2 # D Murphy

3 #

4 # routes.py

5

6 **from** \_source\_code **import** app

7 **from** flask **import** Flask**,** request**,** redirect**,** url\_for**,** render\_template**,**

flash**,** jsonify

8 **from** forms **import** ConversionForm**,** ResultForm**,** EnterForm

9 **from** currencies **import** Currencies

10 **import** json**,** requests

11 **from** decimal **import** Decimal

12

13 user **=** **(**'daithi'**,** 'pass1234'**)** # dummy user for exchange\_api

14

15 api\_convert\_url **=** 'http://localhost:8000/testapi/convert'

16

17 headers **=** **{**'Content-Type'**:** 'application/json'**}**

18

19

20 @app.route**(**'/'**,** methods**=[**'GET'**,** 'POST'**])**

21 **def** home**():**

22 form **=** EnterForm**()**

23

24 **if** request**.**method **==** 'POST'**:**

25 **return** redirect**(**url\_for**(**'convert'**))**

26

27 **elif** request**.**method **==** 'GET'**:**

28 **return** render\_template**(**'home.html'**,** form**=**form**)**

29

30

31 @app.route**(**'/convert'**,** methods**=[**'GET'**,** 'POST'**])**

32 **def** convert**():**

33 currency\_feed **=** Currencies**()**

34

35 currency\_feed**.**test\_feed\_connection**()**

36

37 **if** currency\_feed**.**request\_status\_code **==** 200**:** # case connection

succeeded

38 form **=** ConversionForm**()**

39

40 currency\_dict **=**

currency\_feed**.**get\_currency\_list**()[**'currency\_list'**]**

41

42 currency\_code\_list **=** **[]**

43 **for** currency\_code **in** currency\_dict**:**

44 currency\_code\_list**.**append**((**currency\_code**,** currency\_code **+** '

- ' **+** currency\_dict**[**currency\_code**]))**

45

46 form**.**fromCurrency**.**choices **=** currency\_code\_list

47 form**.**toCurrency**.**choices **=** currency\_code\_list

48

49 **if** request**.**method **==** 'GET'**:**

50 **return** render\_template**(**'conversion.html'**,** form**=**form**)**

51

52 **elif** request**.**method **==** 'POST'**:**

53 **if** form**.**validate**()** **==** **False:**

54 form**.**conversionAmount**.**data **=** ''

55 **return** render\_template**(**'conversion.html'**,** form**=**form**)**

56

57 **else:**

58 from\_currency **=** form**.**fromCurrency**.**data

59 from\_currency\_label **=** currency\_dict**[**from\_currency**]**

60

61 to\_currency **=** form**.**toCurrency**.**data

62 to\_currency\_label **=** currency\_dict**[**to\_currency**]**

63

64 amount **=** form**.**conversionAmount**.**data

65

66 request\_parameters **=** **{**'from\_currency'**:** from\_currency**,**

67 'to\_currency'**:** to\_currency**,**

68 'amount'**:** amount**}**

69

70 json\_request\_parameters **=** json**.**dumps**(**request\_parameters**)**

71

72 json\_response **=** requests**.**post**(**api\_convert\_url**,**

73 json\_request\_parameters**,**

74 headers**=**headers**,**

75 auth**=**user**)**

76

77 **if** json\_response**.**status\_code **==** 403**:**

78 **return** redirect**(**url\_for**(**'unauthorised'**))**

79

80 **else:**

81 result **=** json\_response**.**json**()**

82

83 converted\_amount **=**

round\_to\_two\_places**(**result**[**'converted\_amount'**])**

84 unit\_rate **=**

round\_to\_six\_places**(**result**[**'unit\_rate'**])**

85 formatted\_amount **=** round\_to\_two\_places**(**amount**)**

86 last\_update **=** result**[**'last\_update'**]**

87

88

89 **return** render\_template**(**'conversion.html'**,**

90 form **=** form**,**

91 input\_amount **=**

formatted\_amount**,**

92 from\_currency **=**

from\_currency**,**

93 from\_currency\_label **=**

from\_currency\_label**,**

94 to\_currency **=** to\_currency**,**

95 to\_currency\_label **=**

to\_currency\_label**,**

96 converted\_amount **=**

converted\_amount**,**

97 unit\_rate **=** unit\_rate**,**

98 last\_update **=** last\_update**)**

99

100 **else:** # case connection failed

101 **if** currency\_feed**.**request\_status\_code **==** 403**:**

102 **return** redirect**(**url\_for**(**'unauthorised'**))**

103

104 **elif** currency\_feed**.**request\_status\_code **==** 404**:**

105 **return** redirect**(**url\_for**(**'unavailable'**))**

106

107 **elif** currency\_feed**.**request\_status\_code **==** 'connection\_error'**:**

108 **return** redirect**(**url\_for**(**'unavailable'**))**

109

110

111 @app.route**(**'/unauthorised'**)**

112 **def** unauthorised**():**

113 **return** render\_template**(**'unauthorised.html'**)**

114

115

116 @app.route**(**'/unavailable'**)**

117 **def** unavailable**():**

118 **return** render\_template**(**'api\_unavailable.html'**)**

119

120

121 **def** round\_to\_two\_places**(**amount**):**

122 TWOPLACES **=** Decimal**(**'.01'**)**

123 **return** Decimal**(**amount**).**quantize**(**TWOPLACES**)**

124

125

126 **def** round\_to\_six\_places**(**amount**):**

127 FIVEPLACES **=** Decimal**(**'.000001'**)**

128 **return** Decimal**(**amount**).**quantize**(**FIVEPLACES**)**

currencies.py

1 **from** flask **import** json

2 **import** requests

3 **from** requests**.**exceptions **import** ConnectionError

4

5

6 # set up dummy user for authentication, to match authenticated

7 # user in exchange\_api

8 user **=** **(**'daithi'**,** 'pass1234'**)**

9

10

11 **class** **Currencies():**

12

13 api\_getcurrencies\_url **=**

'http://localhost:8000/testapi/getcurrencies'

14 headers **=** **{**'Content-Type'**:** 'application/json'**}**

15

16 **def** test\_feed\_connection**(**self**):**

17 **try:**

18 currency\_request **=** requests**.**get**(**self**.**api\_getcurrencies\_url**,**

headers**=**self**.**headers**,** auth**=**user**)**

19 **if** currency\_request**.**status\_code **==** 200**:**

20 self**.**request\_status\_code **=** 200

21 **return** **True**

22

23 **else:**

24 **if** currency\_request**.**status\_code **==** 403**:**

25 self**.**request\_status\_code **=** 403

26 **elif** currency\_request**.**status\_code **==** 404**:**

27 self**.**request\_status\_code **=** 404

28 **return** **False**

29

30 **except** ConnectionError **as** error**:**

31 self**.**request\_status\_code **=** 'connection\_error'

32 **return** **False**

33

34 **def** get\_currency\_list**(**self**):**

35 json\_currencies **=** requests**.**get**(**self**.**api\_getcurrencies\_url**,**

headers**=**self**.**headers**,** auth**=**user**)**

36 **return** json\_currencies**.**json**()**

#### Increment #05 code – API

exchange\_api.py

1 # RESTful api (Flask/Python)

2

3 # currency conversion

4 # all data in HTTP methods is JSON format

5

6 # getSpecificRate() and exchange() return floats

7 # getRate() returns a float

8

9 # get\_active\_currency\_list() returns currency

10 # dictionary (JSON format)

11

12 # exchange rates are relative to 1.0 \* Euro

13

14 # code structure adapted from tutorial at

15 # <http://blog.miguelgrinberg.com/post/designing-a-restful-api-with->

python-and-flask

16 # by Miguel Grinberg

17

18 **from** \_source\_code **import** app

19 **from** flask **import** Flask**,** jsonify**,** abort**,** request**,** make\_response**,** json

20 **from** flask**.**ext**.**httpauth **import** HTTPBasicAuth

21 **import** requests

22 **from** rates\_thread **import** RatesThread

23 **from** decimal **import** Decimal

24

25

26 auth **=** HTTPBasicAuth**()**

27

28

29 live\_rates **=** RatesThread**()**

30 live\_rates**.**daemon **=** **True**

31 live\_rates**.**start**()**

32

33 RATE\_INDEX **=** 0

34 LABEL\_INDEX **=** 1

35

36

37 @auth.get\_password

38 **def** get\_password**(**username**):**

39 **if** username **==** 'daithi'**:**

40 **return** 'pass1234'

41 **return** **None**

42

43

44 @auth.error\_handler

45 **def** unauthorized**():**

46 **return** make\_response**(**jsonify**(** **{** 'error'**:** 'Unauthorised access' **}** **),**

403**)**

47

48

49 @app.route**(**'/testapi/convert/<string:currency>'**,** methods **=** **[**'GET'**])**

50 @auth.login\_required

51 **def** get\_rate**(**currency**):**

52 **if** currency **in** live\_rates**.**exchange\_rates**:**

53 **return** jsonify **(** **{** 'rate'**:** live\_rates**.**exchange\_rates**[**currency**]** **}**

**)**

54 **else:**

55 abort**(**404**)**

56

57

58 @app.route**(**'/testapi/convert'**,** methods **=** **[**'POST'**])**

59 @auth.login\_required

60 **def** convert\_amount**():**

61 **if** **not** request**.**json**:**

62 abort**(**400**)**

63

64 **if** **not** 'from\_currency' **in** request**.**json**:**

65 abort**(**400**)**

66

67 **if** **not** 'to\_currency' **in** request**.**json**:**

68 abort**(**400**)**

69

70 **if** **not** 'amount' **in** request**.**json**:**

71 abort**(**400**)**

72

73 **if** **not** validate\_amount**(**request**.**json**[**'amount'**]):**

74 abort**(**400**)**

75

76 **if** request**.**json**[**'from\_currency'**]** **not** **in** live\_rates**.**exchange\_rates**:**

77 abort**(**404**)**

78

79 **if** request**.**json**[**'to\_currency'**]** **not** **in** live\_rates**.**exchange\_rates**:**

80 abort**(**404**)**

81

82

83 converted\_amount **=** exchange**(**request**.**json**[**'from\_currency'**],**

84 request**.**json**[**'to\_currency'**],**

85 request**.**json**[**'amount'**])**

86

87 unit\_rate **=** get\_unit\_rate**(**request**.**json**[**'from\_currency'**],**

88 request**.**json**[**'to\_currency'**])**

89

90 **return** jsonify**(** **{** 'converted\_amount'**:** converted\_amount**,**

91 'unit\_rate'**:** unit\_rate**,**

92 'last\_update'**:**

live\_rates**.**exchange\_rate\_last\_update **}** **)**

93

94

95 @app.route**(**'/testapi/getcurrencies'**,** methods **=** **[**'GET'**])**

96 @auth.login\_required

97 **def** get\_active\_currency\_list**():**

98 currency\_list **=** **{}**

99 **for** currency\_code **in** live\_rates**.**exchange\_rates**:**

100 currency\_list**[**currency\_code**]** **=**

live\_rates**.**exchange\_rates**[**currency\_code**][**LABEL\_INDEX**]**

101

102 **return** jsonify **(** **{** 'currency\_list'**:** currency\_list **}** **)**

103

104

105 @app.errorhandler**(**404**)**

106 **def** not\_found**(**error**):**

107 **return** make\_response**(**jsonify**(** **{** 'error'**:** 'Resource not found' **}**

**),** 404**)**

108

109

110 @app.errorhandler**(**400**)**

111 **def** bad\_request**(**error**):**

112 **return** make\_response**(**jsonify**(** **{** 'error'**:** 'Bad request' **}** **),** 400**)**

113

114

115 **def** get\_unit\_rate**(**from\_currency**,** to\_currency**):**

116 from\_rate **=** live\_rates**.**exchange\_rates**[**from\_currency**][**RATE\_INDEX**]**

117 to\_rate **=** live\_rates**.**exchange\_rates**[**to\_currency**][**RATE\_INDEX**]**

118

119 decimal\_from\_rate **=** Decimal**(**from\_rate**)**

120 decimal\_to\_rate **=** Decimal**(**to\_rate**)**

121

122 decimal\_unit\_rate **=** decimal\_to\_rate **/** decimal\_from\_rate

123

124 # return a string for JSON string

125 **return** str**(**decimal\_unit\_rate**)**

126

127

128 **def** exchange**(**from\_currency**,** to\_currency**,** amount**):**

129 from\_rate **=** live\_rates**.**exchange\_rates**[**from\_currency**][**RATE\_INDEX**]**

130 to\_rate **=** live\_rates**.**exchange\_rates**[**to\_currency**][**RATE\_INDEX**]**

131

132 decimal\_from\_rate **=** Decimal**(**from\_rate**)**

133 decimal\_to\_rate **=** Decimal**(**to\_rate**)**

134 decimal\_amount **=** Decimal**(**amount**)**

135

136 decimal\_converted\_amount **=** decimal\_amount **\*** decimal\_to\_rate **/**

decimal\_from\_rate

137

138 # return a string for JSON string

139 **return** str**(**decimal\_converted\_amount**)**

140

141

142 # validate value of key 'amount' as numerical string

143 **def** validate\_amount**(**amount**):**

144 **try:**

145 float**(**amount**)**

146 **return** **True**

147 **except** ValueError**:**

148 **return** **False**

rates\_thread.py

1 x# class definition for thread object, used by

2 # exchange\_api.py to get live currency rates

3 # from Google feed periodically

4 # (return period = parameter in time.sleep(), measured in seconds)

5

6 **from** flask **import** json

7 **from** threading **import** Thread

8 **import** requests**,** time

9

10

11 **class** **RatesThread** **(**Thread**):**

12 **def** \_\_init\_\_**(**self**):**

13 '''Constructor'''

14 Thread**.**\_\_init\_\_**(**self**)**

15 self**.**exchange\_rates **=** **{**'EUR'**:** **[**'1.00000'**,** 'Euro'**]}**

16

17 **def** run**(**self**):**

18 **while** **True:**

19 json\_live\_rates **=**

requests**.**get**(**'http://spreadsheets.google.com/feeds/list/0Av2v4lMxiJ1AdE9laEZJdzhmMzdmcW90VWNfUTYtM2c/3/public/basic?alt=json'**)**

20 live\_rates **=** json\_live\_rates**.**json**()**

21 self**.**exchange\_rate\_last\_update **=** time**.**ctime**()**

22

23 number\_of\_currencies\_in\_feed **=**

len**(**live\_rates**[**'feed'**][**'entry'**])**

24

25 **for** currency\_entry **in** range**(**number\_of\_currencies\_in\_feed**):**

26 currency\_code **=**

live\_rates**[**'feed'**][**'entry'**][**currency\_entry**][**'title'**][**'$t'**]**

27

28 **if** currency\_code **in** self**.**currency\_codes**:**

29 currency\_label **=** self**.**currency\_codes**[**currency\_code**]**

30

31 rate **=** live\_rates**[**'feed'**][**'entry'**]**

**[**currency\_entry**][**'content'**][**'$t'**][**8**:]**

32

33 self**.**exchange\_rates**[**currency\_code**]** **=** **[**rate**,**

currency\_label**]**

34

35 time**.**sleep**(**300**)**

36

37 currency\_codes **=** **{** 'EUR'**:** 'Euro'**,**

38 'USD'**:** 'US Dollar'**,**

39 'JPY'**:** 'Japanese Yen'**,**

40 'GBP'**:** 'British Pound'**,**

41 'CHF'**:** 'Swiss Franc'**,**

42 'AUD'**:** 'Australian Dollar'**,**

43 'CAD'**:** 'Canadian Dollar'**,**

44 'SEK'**:** 'Swedish Krona'**,**

45 'HKD'**:** 'Hong Kong Dollar'**,**

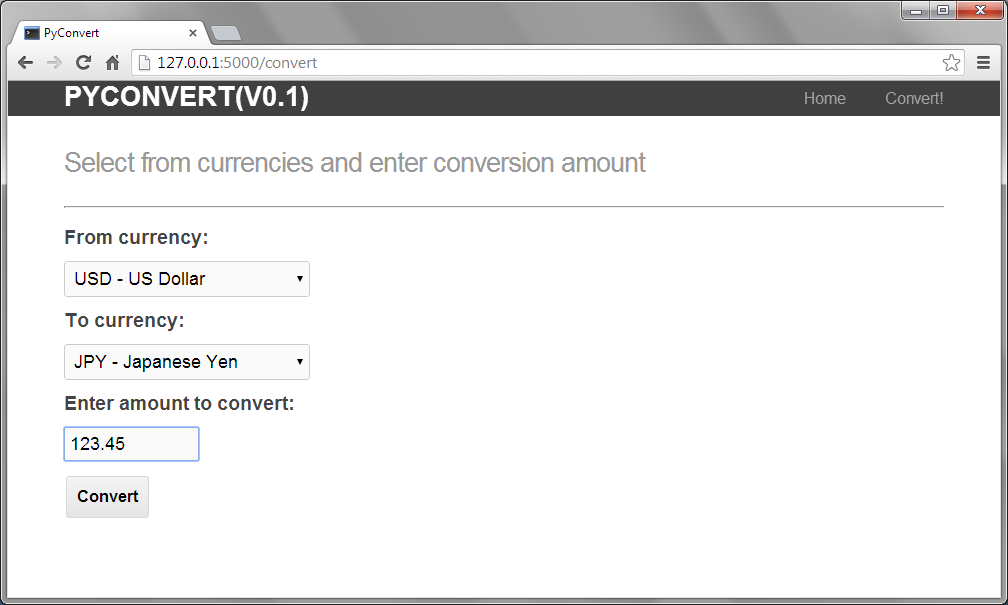
46 'NOK'**:** 'Norwegian Krone'**}**

#### Increment #05 testing

Testing for this increment is identical to that for Increment #02.

#### Increment #05 output

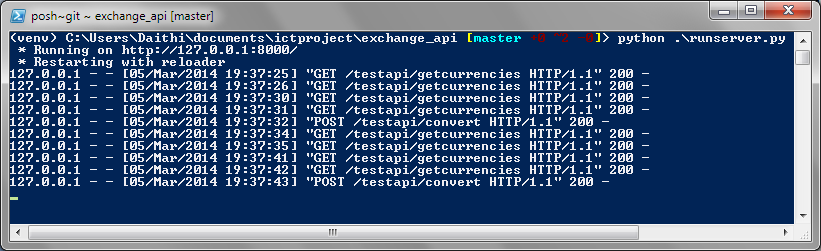
The following figures present a sample execution of the client-side application interacting with the API (hosted locally).



**Figure 8.xx – currency conversion, Increment #05**

****

**Figure 8.xx – currency conversion result with live rates, Increment #05**

****

**Figure 8.xx – Increment #05 API web service running on localhost:8000**

### Increment #06

Increment #06 made some minor changes to the code, including refactoring of the client-side application code, introduction of a configuration file and deployment of the web service from a Windows environment.

#### Client-side code refactoring

After a review of the code for the client-side application it was decided that it would be useful to refactor some of the code to improve efficiency and readability.

The main change involves the elimination of the currencies.py script; in the previous iteration, this script contained a Currencies() class definition, with two associated methods, test\_feed\_connection() and get\_currency\_list(). An instance of this class was created in routes.py, with the test\_feed\_connection() method used to test the connection at the beginning of a request, with a negative response dealt with accordingly. It was decided that a try-except clause would handle this in a more efficient manner.

By employing a try-except clause within the convert() function in routes.py the request proceeds in any case within the try-block (without initially testing to check if the connection exists), and a custom-built exception CurrencyFeedException is raised if the HTTP request response is not equal to 200 (i.e., the response is not an ‘OK’ response). If the exception is raised the associated response code is recorded and the code within the except-block redirects control to the appropriate URL (e.g., ‘API not available’, or ‘User unauthorised’).

In this manner, the code deals with the connection to the API in a neater fashion, and any associated exceptions are handled gracefully.

The relevant extracts from the convert() function in routes.py including the refactored code, and the custom-built CurrencyFeedException exception class are as follows:

33 @app.route**(**'/convert'**,** methods**=[**'GET'**,** 'POST'**])**

34 **def** convert**():**

35 **try:**

36 form **=** ConversionForm**().....**

*#code as previous here....*

104 **except** CurrencyFeedException**:**

105 **if** response\_code **==** 403**:**

106 **return** redirect**(**url\_for**(**'unauthorised'**))**

107

108 **elif** response\_code **==** 404**:**

109 **return** redirect**(**url\_for**(**'unavailable'**))**

110

111 **elif** response\_code **==** 503**:**

112 **return** redirect**(**url\_for**(**'unavailable'**))**

113

114 **else:**

115 **return** redirect**(**url\_for**(**'unavailable'**))**

*#........*

137 **class** **CurrencyFeedException(**Exception**):**

138 **pass**

#### Configuration file

A configuration file was built in this iteration in order to set various application-specific variables. The variable settings are contained in config\_settings.ini, located in the root directory. At present, the file sets values for username and password (for HTTP authorisation), and URL addresses for the web-service.

In order to parse the configuration file a Python extension, ConfigObj, is imported to the application. This extension parses the file and makes values in the configuration file available to the application. In this manner, any such application configuration variables can be changed in this single location.

The configuration file, config\_settings.ini is as follows:

1 [userinfo]

2 username **=** daithi

3 password **=** pass1234

4

5 [urls]

6 url\_getcurrencies **=** <http://exchange-api->

eventlet.herokuapp.com/testapi/getcurrencies

7 url\_convert **=** http://exchange-api-eventlet.herokuapp.com/testapi/convert

#### Deployment of API web-service to Heroku from a Windows environment

The RESTful web-service application was previously deployed to Heroku from a Ubuntu environment using the *Gunicorn* web server. In this iteration, the application was pushed to Heroku from a Windows environment using the *Eventlet* web server for Python.

A minor amendment was required in the runserver.py boot file in order to run the application on an Eventlet web server. Heroku assigns a random port number to the application when it is deployed, and this is set in the ‘PORT’ Python environment variable. The web server is required to listen on this port from the application framework in Heroku. The code included in the boot file (now called runserver\_eventlet.py for clarity) assigns the port in the Eventlet web server to whatever port Heroku has randomly assigned to the PORT environment variable. If the PORT environment variable is not available, the server listens on port 5000 (i.e., in development mode).

The code in runserver\_eventlet.py is as follows:

1 **import** os

2 **import** eventlet

3 **from** eventlet **import** wsgi

4 **from** \_source\_code **import** app

5

6 port **=** int**(**os**.**environ**.**get**(**"PORT"**,** 5000**))**

7 wsgi**.**server**(**eventlet**.**listen**((**'0.0.0.0'**,** port**)),** app**)**

## Final Implementation

## Implementation Conclusions

# Project Conclusions

# References

# Bibliography

1. Telles, M. 2008, p.1 [↑](#footnote-ref-2)
2. http://en.wikipedia.org/wiki/Python\_(programming\_language) [↑](#footnote-ref-3)
3. http://www.python.org/about/ [↑](#footnote-ref-4)
4. http://www.python.org/about/apps/ [↑](#footnote-ref-5)
5. http://www.python.org/psf/ [↑](#footnote-ref-6)
6. http://www.python.org/community/ [↑](#footnote-ref-7)
7. Telles, M. 2008 p.3 [↑](#footnote-ref-8)
8. http://en.wikipedia.org/wiki/History\_of\_Python [↑](#footnote-ref-9)
9. http://docs.python.org/2/whatsnew/2.0.html [↑](#footnote-ref-10)
10. http://www.python.org/download/releases/2.7/ [↑](#footnote-ref-11)
11. http://en.wikipedia.org/wiki/Python\_(programming\_language) [↑](#footnote-ref-12)
12. http://www.python.org/dev/peps/pep-0020/ [↑](#footnote-ref-13)
13. Telles, M. 2008 p.8 [↑](#footnote-ref-14)
14. http://www.diveintopython.net/getting\_to\_know\_python/everything\_is\_an\_object.html [↑](#footnote-ref-15)
15. http://docs.python.org/2/howto/functional.html [↑](#footnote-ref-16)
16. http://en.wikipedia.org/wiki/Python\_syntax\_and\_semantics [↑](#footnote-ref-17)
17. Telles, M. 2008 p.34 [↑](#footnote-ref-18)
18. http://docs.python.org/2/tutorial/controlflow.html#default-argument-values [↑](#footnote-ref-19)
19. Telles, M. 2008 p.209 [↑](#footnote-ref-20)
20. Telles, M. 2008 p.213 [↑](#footnote-ref-21)
21. Telles, M. 2008 p.215 [↑](#footnote-ref-22)
22. Telles, M. 2008 p.226 [↑](#footnote-ref-23)
23. Telles, M. 2008 p.229 [↑](#footnote-ref-24)
24. Telles, M. 2008 p.237 [↑](#footnote-ref-25)
25. https://devcenter.heroku.com/articles/quickstart [↑](#footnote-ref-26)
26. https://devcenter.heroku.com/articles/how-heroku-works#defining-an-application [↑](#footnote-ref-27)
27. https://devcenter.heroku.com/articles/python-faq#do-python-applications-run-behind-nginx [↑](#footnote-ref-28)
28. https://devcenter.heroku.com/articles/how-heroku-works#running-applications-on-dynos [↑](#footnote-ref-29)
29. http://flask.pocoo.org/docs/patterns/packages/ [↑](#footnote-ref-30)
30. http://wtforms.readthedocs.org/en/latest/validators.html [↑](#footnote-ref-31)
31. https://wiki.archlinux.org/index.php/Python\_VirtualEnv [↑](#footnote-ref-32)