RMIT University

Scripting Language Programming Assignment 2

Study period 1, 2018

Version 5

5.6
Examples are labelled as examples.

5.5
Removed description of parameters section in the example of data flows.

5.4
Clarified that flows have to run once a minute.
Corrected double quoting in one of the examples.
Modified “returns” to “prints” in discussion of services.
Harmonised with the marking guide.

5.3
Modified the description of the submission format to put <> around testfiles for part A.
Removed “parameters” component from Data passing format.
Noted which parts of config files are mandatory.

5.2
Time format specification made for part 2

5.1
Clarified parameter passing to services

5.0
Removed configuration parameters from the twitter example. It will no longer be used.

Version 4

Includes “what to submit” instructions.

Marks: 20%

Due date: 27th May 2018

Assessment

The assignment will be based on the overall quality of your work, submission as well as its functionality.
Part 1 9%
Part 2 9%
Quality –2..+2%
Submission –2..0%

i.e. you can lose marks if your work is not of a moderate level of professionalism.

Overview

This assignment will behave in a manner similar to internet tools such as IFTTT and Stringify - a way to specify connections between data sources, behaviours and actions.

You will be required to build an application in Python 3 that will process the configuration of the components, and cause the actions to occur.

The assignment will consist of two parts, as well as a small bonus section.

To simulate the experience in industry the first part will be released initially and then the second part - a change of specifications - will be released later. Making your code robust in the face of potential changes is an important skill that you should practice.

You will be expected to submit both parts of the assignment.

Specifications

Part 1

The program will read a json configuration file specified on the command line, or if one is not presented, from a file named `ifttt.json`. If the configuration file is invalid json, or is missing mandatory sections the program should abort with a suitable error message describing the line the error was found on, if possible.

The configuration file will consist of a dictionary with two entries - a dictionary of services and an array of flows.

A `service` is a named dictionary which contains the program to be run, along with any needed parameters and configuration information.

A `flow` is a named array of services to be called in sequence, with any output from one being inserted as the input to the next service.

For example someone may create a flow description as shown below.

```
{
    "services":{
        "time of day":{
            "program":"date",
            "parameters":"+%d/%m/%Y %H:%M:%S",
        },
        "append to file":{
            "program":"./append.py"
            "parameters":"times.txt"
        },
        "twitter":{
            "program":"./send_to_twitter.py",
        },
    },
    "flows": {
      "Append time of day":[
         "time of day",
         "append to file"
      ]
   }
}
```

Services

A service refers to a program that is available from the command line. Some services may need parameters. These should be provided as command line arguments to the program.

The “append to file” service would be called as…

```
./append.py "times.txt"
```

Parameters can refer to the standard input text via the special symbol `$$`. If found in the parameters the standard input for the service is substituted.

For example someone may create a service description as shown below.

```
      "echo":{
         "program":"echo",
         "parameters":"input is $$"
      },
```

Flows

In the example above the single flow “Append time of day” will be continually run, and will call the services “time of day” then “append to file”. Those services in turn invoke the referenced program.

Each service should return a status value (0 for success, non zero for failure). If a non zero value is returned the flow should finish.

The program should be continually process all the flows, starting each flow no more frequently than once per second (it should attempt once per second though).
Update As discussed in chat sessions and in postings this timing is very hard to achieve, and has been modified to be once per minute. The set of flows should start at the start of every minute if possible.

Services you should create

Services can be programmed in any language, but for this assignment you should use either shell scripting commands (bash) or python. Note that “input” described below is standard input.

Morning

prints “true” if the input is a time prior to 12pm, otherwise “false”. Exits if invalid time.

Afternoon

prints “true” if the input is a time between 12pm and 6pm, otherwise “false”. Exists if invalid time.

Evening

prints “true” if the input is a time after (past 6pm), otherwise “false”. Exists if invalid time.

Quit if false

exits (non zero value) if the input is “false”

Not

negates the input - True to False, False to True. Exits if not one of those values.

Challenge : Can you construct…?

A service that only continues if run on the hour? Can you generalise this to allow for every n minutes, or every n seconds?

A service that returns sunrise time, another that returns sunset (using https://sunrise-sunset.org/api)

A flow that saves the time to a file every hour, but only in the evening?

Part 2

Overview

Part 1 facilitated communication between flows by passing plain text from the output of one service to the input of the next.
As well all flows and services are currently described in a single file (ifttt.json by default).

Part 2 will be a modification of part 1 with data being passed from one service to another via specially constructed json messages.

As well each service is to be stored in an individual directory whose name is that of the service.
Inside the directory is a config.json file containing a description of the input and output formats of the data it consumes and produces.

Flows are to continue to be stored in a single file (specfied on the command line, or `ifttt.json` by default).

The flow file can be edited while the service is running, and the program should check it every time through its main processing loop to see if it has changed. If so it should reload the file and run the new flows.

Services can be installed at run time (i.e. a new directory is with a config.json file is installed, with an option python program), or they may be removed if a flow is edited (i.e. they go with the flow).

The configuration file format is loosely based on the concepts that can be found in Mozilla's iot api.
The actual format will be described later.

Configuration file format

Each service has a configuration file (`config.json`)

For example someone may create a configuration file as shown below.

```
{
  "name":"Evening",
  "program":"./evening.py",
  "description":"Returns true when the input time is between 6pm and 12a
m",
  "configuration" : {
    "any" : "key value pairs you need"
  },
  "parameters":"any parameters you need for the program",

  "input" : {
    "type": "time",
  },
  "output": {
    "type": "boolean"
  }
}
```

The format describes the values that can be passed into and out of service.
Valid options for type are

- time (format HH:MM:SS, 24 hour time)
- boolean
- number
- string
- dictionary
- array

These mostly correspond to the main types found in json. These types should be verfied when a flow is running.
The parameters section allows you to specify any values that the program may need.
The mandatory fields are **name**, **program** and **description**.

Parameters and configuration key value pairs are passed in as command line arguments…

```
./evening.py "any parameters you need for the program" -any "key value pairs you need"
```

Data passing format

Data passed between services is represented as a json structure (as a string).
This is structured as shown in the example below.

```
  {
    "data"  :data as specified in the input/output sections of config for the producing service (if present)
  }
```

For example the evening service may expect data in the format

```
  {
    "data" : "10:03:10",
  }
```

Similarly it might output data in the format

```
  {
    "data": true
  }
```

The program should ensure that the data sent to a service and produced by a service is in the correct format if the input/output fields in a services' config.json are present.

Python modules/functions you'll need

Calling a program

The following code is needed to call an external program

```
from subprocess import run, PIPE

command = ['date', '+%d/%m/%Y %H:%M:%S']
p = run(command, stdout=PIPE, input='', encoding='utf-8')
```

```
print(p.returncode)
print(p.stdout)
```

The parameter “input” (a string) will be provided as the standard input for the program. p.returncode is the standard Unix shell value - 0 for success, non zero for failure. In a python program this value can be set when you call sys.exit().

p.stdout contains all of the output made by the service (i.e. anything displayed/printed)

Reading from standard input

If you write a service in Python you may need it to read from standard input.

```
import sys

line = sys.stdin.read()
```

What to submit

You should submit your Part 1 and Part 2 as separate components of the same submission.

The format of your submission must follow the following layout

```
partA
        ifttt.py
        ifttt.json
        <other python services you have created>
        <testfiles>


partB
        ifttt.py
        ifttt.json
        <directory for each service>
            config.json
            <executable python program>
        <testfiles directory>
```

The naming conventions should be followed.
Your directories should not be enclosed in any other directories.

Testing

Your program will be tested with a range of different configuration files than the one example provided above.

Ensure you create a range of different configuration files to test your program with.

Resources

json formatter is a useful tool for verying and formatting your json.