Red Hat Python Lab - Lesson 1: Introduction to Python

Daniel Mach <dmach@redhat.com>
Martin Sivák <msivak@redhat.com>

Useful links

- Python Quick Reference: http://rgruet.free.fr/#QuickRef
- Python docs: http://www.python.org/doc/
- PEP 8: http://www.python.org/dev/peps/pep-0008/
- pep8.py: http://pypi.python.org/pypi/pep8/
- pylint: http://www.logilab.org/project/pylint
- Epydoc: http://epydoc.sourceforge.net/

Interesting links:

- Letajici cirkus on root.cz: http://www.root.cz/serialy/letajici-cirkus/
- py.cz: http://www.py.cz/
- Unladen swallow: http://code.google.com/p/unladen-swallow/
- Python in Python: http://pypy.org/

Don't forget

- Batteries included -> read docs, find existing function, use it
- Follow PEP 8
- Pylint is your friend
- Write unit tests
- Objects are not everything, don't make your code over-complicated
- Return from functions as soon as possible -> more readable code, better indentation
- import this - python "cornerstones"

Python features

- Object oriented (programming, scripting) language
- Weakly-typed language (it has types; type of a variable is defined on assigining a value)
- Uses indentation to define code blocks (use 4 spaces)
- Tends to be backward compatible
- Has bytecode (faster than ruby, perl)

Types

All types are fake objects - they are not real objects, but they have methods

- str: x = "text" # *immutable*
- unicode: x = u"text" # *immutable*
- int: x = 1 # *immutable*
- long: x = 11 # *immutable, int is automatically converted to long when it reaches it's limit*
- float: x = 1.0 # *immutable*
- bool: x = True, x = False # *immutable*
- NoneType: x = None # *immutable*
- list: x = [1, 2, 3] # *mutable*
- tuple: x = (1, 2, 3) # *immutable*
- dict: x = {1: "foo", 2: "bar"} # *mutable, key=value*
- set: x = set([1, 2, 3]) # *mutable*

```
# comprehensive help
help(str)

# list all attributes of a variable
dir(str)
dir("a string")
```

Types - strings

- Immutable - changing means copying to a new one
- Normal strings vs. unicode
- Multiline definitions
- Formatting

```
# initialization
s1 = "string"
s2 = u"unicode string"
s3 = "%d bottles of %s wine" % (99, "red")
s4 = "%(number)d bottles of %(colour)s wine" % {"number": 99, "colour": "red"}
s5 = """long string spanning
multiple lines with "quotes"
"""

# basic operations
s3.split()
s3.split(" ", 1)
s4.upper() != s4.lower()
", ".join(["apples", "oranges", "coconuts"])
s4.replace("o", "a", 1) + "yard"
```

Types - lists, tuples

- Mutable list vs. immutable tuple
- Sorted lists of values
- Access via index

```
list1 = [1, 2]
for i in list1:
    print i
# 1
# 2

list1.append(3)
print list1
# [1, 2, 3]


tuple1 = ("a", "b")
for index, i in enumerate(tuple1):
    print index, i
# 0 a
# 1 b
```

Types - dicts

- Dictionary is a basic Python structure, used for almost everything
- Maps keys to their values
- dict.items(), dict.values() and dict.keys() vs. dict.iteritems(), dict.itervalues() and dict.iterkeys()
- key in dict vs. dict.has_key(key) # *don't use has_key() - it was dropped in Python 3*

```
# comprehensive help
help(dict)

# creating an empty dictionary
d1 = dict() # dict(key=value, ...)
d2 = {}     # {key: value, ...}

# assigning a value to key
d1["key"] = ["1", 2, 4, 5]
d2[12587] = "value"

# setting a default value
d2.setdefault("foo", "spam")
d2.setdefault("foo", "eggs")
# d2["foo"] contains "spam" (2nd setdefault doesn't do anything)

# getting value back
print d1["key"]
print d2.get("key")
print d2.get("key", "default value")

if "key" in d1:
    print "'key' found in dict 'd1', value is: %s" % d1["key"]

# iterating over dictionary
for k,v in d2.iteritems():
    print k, "=", v
```

Function arguments

```
def write(msg, error_code=0):
    print locals()
write("x")
# {"msg": "x", "error_code": 0}


def write(msg, error_code=0, exception_info):
    print locals()
write("x", "some information about the exception")
# SyntaxError: non-default argument follows default argument


def write(msg, error_code=0, *args, **kwargs):
    print locals()
write("x", "y", "z", info="some information about the exception", error_code=1)
# {"msg": "x", "error_code": 1, args=["y", z"], kwargs={"info": "some information about the exception"}}
```

Workflow

- Analyze what needs to be done
- Think about interfaces, functions, objects, modules
- Write a unit test (interfaces!)
- Start writing a program

Project: IRCBot

Program runs in an endless loop:

- Read input from keyboard
- Parse and process it
- Print results to stdout
- Handle Ctrl-C correctly

Read input from keyboard

Do not start coding! We're only defining interfaces at this moment.

```python
#!/usr/bin/python
def read():
    """
    Read input and return it.

    @return: read data
    @rtype: str
    """
```

Write output

```
def write(arg):
    """
    Write (send) argument to output.

    @param arg: text to be written to the output
    @type arg: str
    """
```

Endless loop

```
if __name__ == "__main__":
    while True:
        try:
            msg = read()
            write(msg)
        except (KeyboardInterrupt, EOFError):
            break
```

Command and text processing

There will be two different kinds of test processing in the bot:

- One will be called command and will be activated by it's designated keyword appearing at the beginning of user input. (> *shutdown*<*enter*> or > *calc 1+1*<*enter*>). If the command is recognized, it will cause the bot to print output of the command instead of just copying the user's input.
- The second kind will be text filtering. Filter will watch the input text for sequences appearing anywhere inside the text and will perform actions like replacing some words, counting words, storing some info... All those actions will be mostly transparent to the user and the bot will usually print the user's input as it did before (possibly with some words replaced).

the API for commands

- Use global dict COMMANDS to map {"command": *function*}
- We need to distinguish whether the message was already processed or not so the referenced *function* will return one of:
    - *state.done(output) - command recognized, output returned*
    - *state.next() - command *not* recognized, continue processing*

the API for text processing (FILTERS)

- Use global list FILTERS to get an array of [*function1*, *function2*, *function3*]
- *functionX*'s return values are similar to COMMANDS' functions
    - *state.done(str) - do not process other filters, return str*

- *state.next() - continue processing*
- *state.replace(str) - replace text with filter's result and continue (swear words filter, ...)*

Return codes for command and filter callbacks

To make the API a bit consistent between all of you, we will be using the provided state module (we will learn about them in detail during the next lesson) and set of functions.

```
import state - this prepares the module to use so put it at the beginning of your source ircbot.py file

# return codes to be used in command and filter functions

# this will return a result saying "everything is done and the result is data"
return state.done(data)

# this will return a result saying "I do not know what to do with this, let someone else to try"
return state.next()

# this will return a result saying "I understood that, but lets replace input with new data and behave like next()"
return state.replace(data)

# test functions to get info about the return code, used in parse()
state.is_done(var)
state.is_next(var)
state.is_replace(var)      - these will check what type is the result we have

# if the result vas done or replace, the data can be accessed using value attribute of the result
var.value                  - the returned result's data are accessible in value attribute
```

Parse message

```
def parse(msg):
    """
    Parse a message, return relevant output.

    @param msg: unparsed message
    @type msg: str
    @return: message to print or None (if there's no output to print)
    @rtype: str|None
    """
```

- The *parse(msg)* function reads COMMANDS and FILTERS variables to figure out what has to be done with the user input *msg*. To make sure your function works correctly (including the processing of all the state return codes) we have prepared a test file for you. It requires that your project is called ircbot.py and you have state.py in the same directory. If you execute it, it will test your code and reports all the errors in your implementation of *parse(msg)*.

command block

- get a first word (*cmd*) from *msg* and test if it is a valid command (*if cmd in COMMANDS:*)
- if yes: execute the corresponding function (*ret = COMMANDS[cmd](rest_of_the_input)*) and process the result *ret* (*if is_done(ret):*, *if is_next(ret):*)
- if the result was state.done, return provided value *ret.value*
- if the result was state.next or it wasn't a command, do nothing and continue to the following FILTERS block.

filters block

- iterate over FILTERS (*for f in FILTERS:*) and call each function from it, store the result to *ret*. (*ret = f(msg)*)

- if the result was state.done (*if state.is_done(ret):*), return provided value (*return ret.value*)
- if the result was state.replace (*if state.is_replace(ret):*), replate text you are working with with provided value (*msg = ret.value*)
- execute next function from FILTERS

Command: Shutdown

- Parser handles 'SHUTDOWN' command
- This command raises SystemExit exception
- Exception must be properly handled

Command: Calculator

- Implement a simple calculator
- Example: = 1 + 2 * (3 + 4)
- You can use 'eval', but make sure that the input is sane (eval executes anything it gets and can run any arbitrary code)

Command and filter: Word-count

- Use global variable WORDCOUNT to store number of all words bot "receives" (tip: implement a filter)
- Command 'word-count' displays: Actual word count is X words.

Command and filter: Karma

- *We will focus on command & filter Karma during next lesson*
- Use global dictionary KARMA to store karma points
- foo++, foo-- increases or decreases karma points
- When karma points == 0, remove record from KARMA
- Command 'karma foo' displays: 'foo' has X points of karma.

Unit test

- Write 1 unit test class for each feature
- Write a test method for each use case
- Think about extreme data (0, +1, -1, big values, different type, uninitialized data)

```
import unittest
import ircbot

class TestParseKarma(unittest.TestCase):
    def setUp(self):
        # prepare test environment to start from scratch in each test
        ircbot.KARMA = {}

    def tearDown(self):
        # clean things up
        pass

    def test_FOO(self):
        ircbot.parse("foo++")
        self.assertEqual(ircbot.KARMA["foo"], 1)
        self.assert_("foo" in ircbot.KARMA)
        self.assertRaises(KeyError, ircbot.KARMA.__getitem__, "bar")


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

Next lesson

- Karma command and filter
- Objects