

The Python:



Welcome to the first Python: Rag.

This newsletter is a way of publishing any article you would care to write related to the Python programming language - it's formatted as a pdf document to be easily printed and left lying around.

So please contribute! Neither formality nor rigour are required - we'll shove just about anything in.

The license is creative commons - so it's free to print and distribute, we particularly welcome beginner articles - for and by beginners, so don't feel you need to be an expert.

The newsletter is community produced - payment cannot be made for any articles offered, and you should indicate your willingness for the article to be published under the Creative Commons license.

News

Python 3.1 is out - see www.python.org

Europython took place in Birmingham, uk and was a sort of heaven for python enthusiasts, see www.europython.eu

Next year's is also in Birmingham - be there!

```
>>> import datetime
>>> datetime.date.today().strftime("%B %Y")
'July 2009'
```

Contents

Today's Tip

Unpacking in a for loop, and whats with the 'is'?

Monthly Module

The ConfigParser module is given a going-over.

Beginners Bits

The while loop, poor relation to 'for'.

Aardvark Article

The Search for the Holy GUI

ATTRIBUTION

www.pythonrag.org

The Python: Rag is a monthly newsletter covering any aspect of the Python programming language.

If you wish to contribute - send your text to editor@pythonrag.org

The topic should have some relationship to python, the style can be anything, code and images can be included. The level of python knowledge can be beginner to brain bashing.

Your article may not be published if (in our opinion) - it is completely nutty.

Any opinions expressed in these articles are those of the authors only, and not the producers of The Python: Rag.

Today's Tip

Very often I get a blind spot and do something in a way that isn't the best - then someone comes along and shows me a neater way.

That's great - but then I'm annoyed my previous code is not as good as it should be. It happened recently; I very frequently have lists of lists, and have to go through them in a for loop - and need access to the sub lists elements.

Easy enough - the elements of each sublist are accessible by the normal 'listname[index]' - but then I have a lot of square brackets and numbers and have to keep track of which number refers to which list element.

Then I found you can unpack the sublists, like:

```
>>> x=[[1,2,3], [4,5,6]]
>>> for a,b,c in x:
...     print a
...     print b
...     print c
...
1
2
3
4
5
6
```

And this is great - I can now use descriptive variable names for the elements.

And now for something completely different:

What's going on here?

How come 'p is q' but not 'a is b' - and does it matter? Who cares? Send your opinions in, and they may be published if they are:

- a) clean
- b) refer to the topic in some way
- c) Don't refer to the topic - but are interesting anyway.

```
>>> a = 500
>>> b = 500
>>> a == b
True
>>> a is b
False
>>> p = 50
>>> q = 50
>>> p == q
True
>>> p is q
True
```

Monthly Module - ConfigParser

This article is an example of how I use the module ConfigParser, it is not necessarily the best or correct way - its just an example of how I use it. ConfigParser is also capable of more than I have shown here.

The ConfigParser module reads and writes configuration files, of the format shown on the left.

In other words, a list of parameter strings, with equal signs, pointing to strings of values.

And these parameters optionally split into sections.

Currently I have not written anything large enough to need the sections - however I generally still place parameters in sections, just to aid readability - so parameters with an associated function are grouped together.

One way I like to use ConfigParser within my programs, is to initially create a dictionary of configuration parameters, set with initial values, something like that shown on the left:

So this dictionary, and its default values are set when this particular module is first run.

```
[web]
port="8000"
browser_refresh="90"

[service]
service_name="Railway Timetable"
poll_interval="300"
log_records="50"
```

```
import ConfigParser
import sys
import os

my_cfg["PORT"]="8000"
my_cfg["BROWSER_REFRESH"]="90"
my_cfg["SERVICE_NAME"]="Railway Timetable"
my_cfg["POLL_INTERVAL"]="300"
my_cfg["LOG_RECORDS"]="50"
```

Note, all values are strings, if within the program I need integers, I have to convert them.

You may note that I capitalize, and don't put spaces in the index names, this is my own preference.

I then usually write code to:

- * Set the config file name and filepath.
- * A function to read the config file, importing in to my dictionary any values set in the config file, but if any parameters are not set, then the above defaults are used.
- * A function to write the config file, updating it with any parameters that require changing.

I may also create several functions to check values to ensure they are reasonable - necessary if there is any chance the config file may be hand edited.

ConfigParser continued ..

So to set the name and location of the config file, I could put it in the same directory as my script, as follows:

```
# The directory where the script is kept
SCRIPTDDIRECTORY=os.path.abspath(os.path.dirname(sys.argv[0]))

# The filename is "myconfig.cfg", and is held in the script directory
config_filepath=os.path.join(SCRIPTDDIRECTORY, "myconfig.cfg")
```

So config_filepath is the full path and filename where the config file is to be kept.

To read the config file:

The following is an example of a function to read the config file, and place the values in the my_cfg dictionary. As the dictionary is already populated with default values, any values not read, will already be set with their defaults.

```
def readconfig(my_cfg, config_filepath):

    # Create a ConfigParser object, to read the config file
    cfg=ConfigParser.ConfigParser()
    fp=open(config_filepath, "rb")
    cfg.readfp(fp)
    fp.close()

    # Read each parameter in turn

    # Web section values
    if cfg.has_option("web", "port"):
        my_cfg["PORT"]=cfg.get("web", "port")
    if cfg.has_option("web", "browser_refresh"):
        my_cfg["BROWSER_REFRESH"]=cfg.get("web", "browser_refresh")

    # service section values
    if cfg.has_option("service", "service_name"):
        my_cfg["SERVICE_NAME"]=cfg.get("service", "service_name")
    if cfg.has_option("service", "poll_interval"):
        my_cfg["POLL_INTERVAL"]=cfg.get("service", "poll_interval")
    if cfg.has_option("service", "log_records"):
        my_cfg["LOG_RECORDS"]=cfg.get("service", "log_records")

    return
```

ConfigParser continued ..

In a real application there would be some error checking - both to check the config file exists, and also instead of just assigning each dictionary value to the value read from the file - perhaps a function call to validate the input would be necessary.

To write the config file:

And the following function writes the dictionary of configuration parameters back to the config file.

```
def writeconfig(my_cfg, config_filepath):

    cfg=ConfigParser.ConfigParser()
    cfg.add_section("web")
    cfg.add_section("service")

    cfg.set("web", "port", my_cfg["PORT"])
    cfg.set("web", "browser_refresh", my_cfg["BROWSER_REFRESH"])

    cfg.set("service", "service_name", my_cfg["SERVICE_NAME"])
    cfg.set("service", "poll_interval", my_cfg["POLL_INTERVAL"])
    cfg.set("service", "log_records", my_cfg["LOG_RECORDS"])

    fp=open(config_filepath, "wb")
    cfg.write(fp)
    fp.close()

    return
```

So, calling readconfig would read the variables into your dictionary, and at any time, writeconfig would write them back out to the config file.

The ConfigParser module can do more - including nested sections - but if you want to find more; that's what the manual is for. I hope you've found this taster interesting.

Beginners Bits - the while loop

If you want to do some repeated calculation and then stop when a certain condition is met - the 'while' loop is your man.

The code to the right starts with an x value of 3 and then keeps printing x, and incrementing it by 1 each time through the loop.

As soon as x reaches 8 the loop will end, so this program will give the following output:

```
3
4
5
6
7
now out of loop
```

If the condition of the 'while' statement evaluates to True, then the loop keeps running. So a statement such as while True: could keep running until you take a hammer to your PC.

However there is a way out - the 'break' statement jumps out of the loop, and could be used with an 'if' condition as shown to the right. Similarly a 'return' (if the loop is in a function definition), will break out and return from the function call.

In each of the two examples given - the question remains - what is the value of x once the loop has been left? That is, what would print x give, if it were placed after the print "now out of loop" statement?

In the first case, x would be 8 - as that is what caused the while condition to fail, the value 8 was not printed in the output given above since the program flow started from the end of the loop once the loop condition was False - and so the print statement was never run with x equal to 8.

In the second case x would be 45 - this caused the 'break' command to run.

So the while statement has its place - though in most programs the 'for' loop is used more frequently, as looping through sequences (which is what the for loop does) - is a very common requirement.

```
x = 3
while x < 8:
    print x
    x += 1

print "now out of loop"
```

*The 'x += 1' adds one to x
it is short for x = x+1 and can be used with
other values, for example
x += 3 is equivalent to
x = x+3*

*The 'break' statement can be used to force a
jump out of the loop:*

```
x = 3
while True:
    print x
    if x == 45:
        break:
    x += 1

print "now out of loop"
```

*The 'break' statement also works with 'for'
loops.*

The Search for the Holy GUI

story = ""

And there came a time when I knew I had to face the terrible truth - my application needed a GUI Framework.

I did as we all do - tried a few out, and hit that dark level of despair when you know, that no matter which you choose - it will be the wrong one, and it will never, ever have that most ancient of all functions - a reasonable print mechanism.

So learning that he existed, I sought him out - that legend amongst men, known only as the Python Guru, who lived alone in a cave on a lonely Scottish island where the internet did not prevail, for he had turned his back on his old life.

It took time - those who knew where he was would say nothing, those who did not know would give unwanted advice, and strangest of all - those fountains of knowledge, Google and Wikipedia were eerily silent.

But I persevered, and eventually followed the barely visible track up the hillside, until I came to the cave - and there in its opening, sat cross-legged in meditation was the Guru himself. Dressed only in Bermuda shorts, his torso was tattooed with writhing Pythons, his beard reached to his navel. He saw me coming, but was silent.

I stood before him and spoke - "Forgive my intrusion oh holy Guru, but I need to know - which is the best GUI Framework for my application?"

He sighed, and asked, "Oh my son, why do you need this knowledge?"

I bowed my head humbly, and answered. "It is fundamental, for my application will be utterly reliant upon it, a bug in it will be a bug in my application that I can do nothing about, I will be at the mercy of its developers, and to get any changes done I will have to crawl on my knees to them and kiss their arses."

The Guru looked upon me with sympathy, and his last words to me - which I shall never forget were:

"My son, you must first know the fundamental truth, all the GUI Frameworks are shi...."

But at that moment, in mid word - a crackling discharge of energy burst around him, and to my astonishment, he sat frozen, unmoving, caught like a living snapshot. Behind me I heard two voices, saying in unison:

"Hail to the forces of Chaos!"

I turned and was amazed at the sight; two men, dressed in some sort of metallic armour, their faces covered with dark visors, holding unearthly weapons which still glowed from the recent discharge.

"Who are you?" I asked, "and what have you done to the Guru?"

"We are Time Agents, from the far future. We have suspended the Guru in a time bubble, for the world is not yet ready for what he was about to reveal - for if the best GUI Framework became known and was universally implemented, the consequences would be dire indeed, altering the future and our very existence."

I think back in pride of my response, I was angry and defiant: "You will not succeed in suppressing this knowledge, in the name of the Guru I will compare every line, method and performance indicator of every GUI Framework, I will take huge user surveys and compile massive spreadsheets - I will devote my life to it, if necessary, but I will discover the best!"

One of the time agents lifted his visor, and looked upon me with a strange expression, that might have been sadness, "I am truly sorry," he said, "but we cannot allow it."

At that moment, they both adjusted their weapons, clicking some new devilry into place, I turned from them and began to run - even as I felt strange energies play about me, and a burning sensation in my very soul.

They did not succeed in encapsulating me in a time bubble - but even as I fled, a new inspiration struck me. The GUI Frameworks I had investigated were not worthy, I would solve the problem myself - I would write a newer, greater, better GUI Framework. One that would gather disciples from across the world, that had non of the failures of the miserable efforts that existed now.

For I knew, with an absolute conviction; The World Needs another GUI Framework!

""""

For variations

```
variation1 = story.replace("GUI Framework", "Web Framework")
variation2 = story.replace("GUI Framework", "Programming language")
variation3 = story.replace("GUI Framework", "Virtual Machine")
variation4 = story.replace("GUI Framework", "Package management system")
```