Some Simple Python Programs

Contents

1 Skills/Concepts

This module will involve creating some simple, hello-world style Python programs. This will involve the following concepts:

- interpreted languages
- modules/libraries
- functions
- console use

1.1 Interpreted Languages

Python is an interpreted language. Unlike programs that compile into machine code and run directly on the CPU, Python compiles into something that still requires another program to execute. Other examples of interpreted languages: C#, Java, JavaScript, Perl... Some non-interpreted languages are Assembly, C, C++, Go...

1.2 Modules/Libraries

Code, whether compiled or not, is generally organized into statements that are placed into functions, and functions are collected (possibly within objects) into modules or libraries. Python modules are generally in single files or organized hierarchically in directories.

This topic will be revisited later, a lot.

1.3 Functions

Functions are a logical collection of statements that generally have a specific purpose. By collecting statements into a single functional unit, it can be easily re-used. This avoids naively copying and pasting code, but more importantly it reduces duplicate code. Well written functions make it easier to re-use code and solve problems.

Generally, functions have a name, a return type, and take parameters. Python functions are no exception. Examples come later.

1.4 Console Use

This module relies heavily on the command-line. You will be using a terminal emulator to execute the following commands.

Command	Description
git	A version control system
touch	Creates or updates access times
ls	Lists files
chmod	Changes the mode of a file
pydoc	Views Python documentation

2 Preparation

You should be fine booting up any Linux distribution as long as it has Python. Actually, any operating system with a Python interpreter should suffice, but this module will use the Kali live systems we have been using.

3 Create your first Python program

3.1 Background

This module will demonstrate creating some simple Python programs.

3.2 Process

Commands will be entered into a terminal but file editing may be done in any text editor.

1. Create a directory to use as the root of your project and make it a git repository.

```
mkdir pyplay
git init pyplay
```

Successful output should be similar to the following:

```
Initialized empty Git repository in /home/user/pyplay/.git/
```

2. Enter the newly created directory and create a program, we will use touch for this so that certain editors will know right away to apply the appropriate syntax highlighting.

```
cd pyplay
touch first.py
```

3. Make the new file executable so that it can be run by typing its name:

```
chmod +x first.py
```

4. Edit the new file, for example:

```
gedit first.py
```

Or:

```
subl first.py
```

Or:

```
gvim first.py
```

Or:

```
vim first.py
```

or...

5. Put the following text into the file:

```
#!/usr/bin/python

print("Hello, world!")
```

6. Save the file using whatever means your editor uses. (Ctrl+S, :w, etc)
7. Execute your new program from a terminal:

```
./first.py
```

Successful output will look like the following:

```
Hello, World!
```

8. If your program is working, commit it to revision control so you can refer back to it later.

```
git add first.py
git commit -m "Committed working version of my first python program."
```

9. Confirm it was committed by viewing the git status:

```
git status
```

Successful output will look like the following:

```
Author: user <user@example.com>
Date:   Mon Sep 12 21:30:24 2016 -0400

    Committed working version of my first python program.
```

3.3 Conclusion

You've create a simple program using python, but this program might be considered of poor design. Successive modules will improve upon this.

4 Make your first python program marginally better

4.1 Background

It's generally considered a poor practice to put raw python statements into a python program. Instead, code should be put into functions and invoked only when the code is being run as a program. The reason for this is that there is no difference between a python program and a library. So, for maximum reusability all programs should be written as though they are a library.

Another good motivation for this is the tool for viewing python documentation will inadvertantly execute the code in a program when it parses it to display the documentation.

So, in this module, we will add documentation and a main function to our program.

4.2 Process

1. Open `first.py` in your editor of choice and add update it to look more like the following:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

''' Our first python program.

This is how a top-level module or program includes embedded documentation.
'''

def main():
    '''
    This is a function containing the main logic of our first program.  It's
    placed here to avoid accidental execution when viewing documentation or if
    someone else attempts to import this program to use some function contained
    in it.
    '''

    print("Hello, world!")

if __name__ == "__main__":
    main()
```

2. Run your new program:

```
./first.py
```

Successful output will look like the following:

```
Hello, World!
```

3. View the embedded documentation in your program using the pydoc command (note there is no file extension when you do this):

```
pydoc first
```

This will launch a simple command-line utility for viewing python documentation. Successful output will look like the following:

```
Help on module first:

NAME
    first - Our first python program.

FILE
    /home/stephen/pyplay/first.py

DESCRIPTION
    This is how a top-level module or program includes embedded documentation.

FUNCTIONS
    main()
        This is a function containing the main logic of our first program.  It's
        placed here to avoid accidental execution when viewing documentation or if
        someone else attempts to import this program to use some function contained
        in it.

~
~
~
~
(END)
```

4. Commit your new changes:

```
commit -a -m "Improved the program layout and added documentation""
```

Successful output will look like the following:

```
[master d2263cd] Improved the program layout and added documentation
 1 file changed, 17 insertions(+), 1 deletion(-)
 mode change 100644 => 100755 first.py
```

4.3 Conclusion

Although this is overkill for such a simple program, this is a more correct way of structuring your program.

We'll discuss the different changes, but in summary this updated program is a decent example of how to:

- specify the file encoding to avoid conversion errors
- modularize your code for inclusion in other people's programs
- add documentation to your code for use by pydoc
- avoid side effects when viewing your module documentation in pydoc

5 Use git to spot changes

5.1 Background

Now that we have a repository with some meaningful content, we'll simulate introducing a bug and using git to highlight differences for us.

5.2 Process

1. Introduce a bug in the code. Open `first.py` and delete a character from the last line so that instead of saying main it says mai.

2. Use the git diff command to view the edit:

```
git diff
```

Successful output will look like the following:

```
diff --git a/first.py b/first.py
index 3b873ed..c0d2fd2 100755
--- a/first.py
+++ b/first.py
@@ -17,4 +17,4 @@ def main():
     print("Hello, world!")

 if __name__ == "__main__":
-    main()
+    mai()
```

3. This output can be difficul to see and interpret, so lets use something meant for humans and use the difftool command of git:

```
git difftool
```

Successful output varies based on the configured application, but generally there will be a window displaying the old and new version of the file with colors showing where content appears to have been deleted or added, or otherwise modified.

4. Lets discard the changes and go back to the previous version of the file:

```
git checkout first.py
```

5.3 Conclusion

The bug introduced was a pretty trivial example, but it should be apparent that you can, at the very least, view the differences in your working copy of a file versus the repository version. This is useful for hunting down bugs, summarizing changes for a future commit message, or just seeing what's changed. The `git diff` command gives the classic `diff` command output which is great for viewing short changes or creating a patch to share a changeset with others.

The `git difftool` command is better for interactively viewing large sets of changes and the specific tool it invokes depends on how the system is configured.