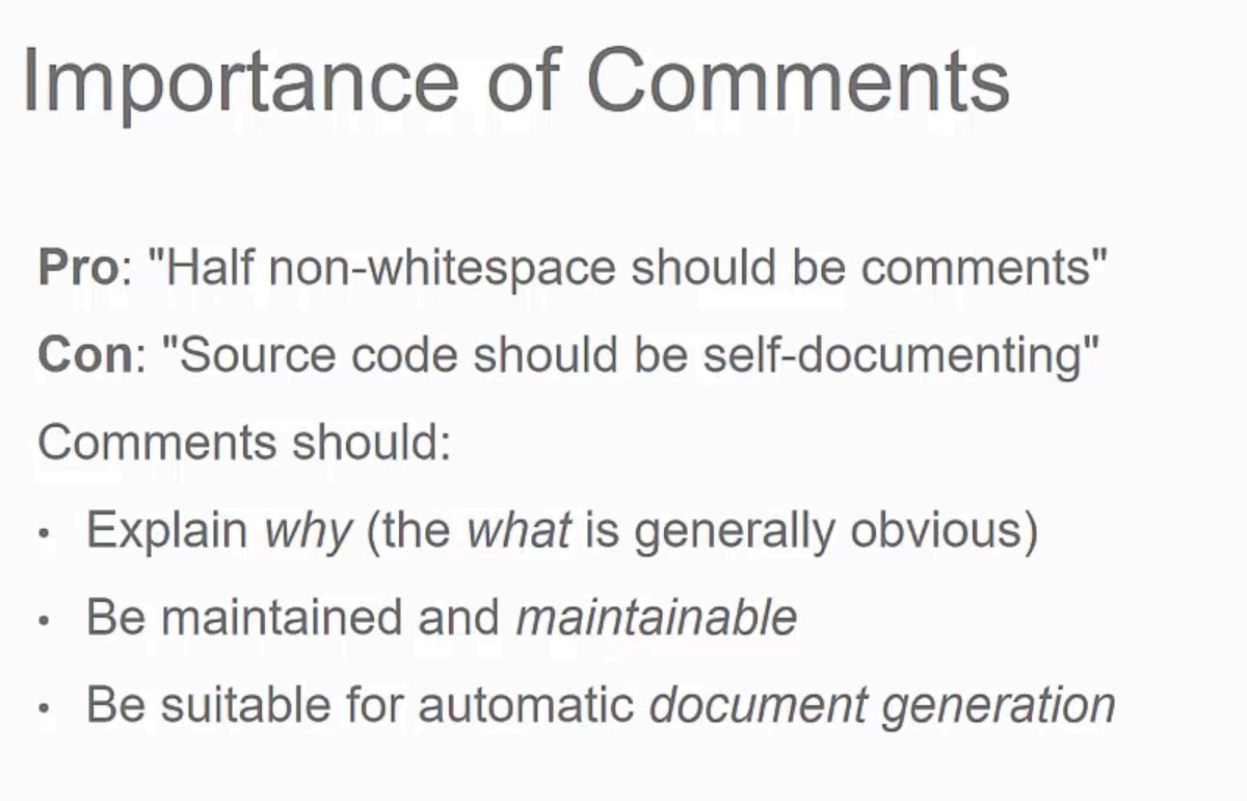
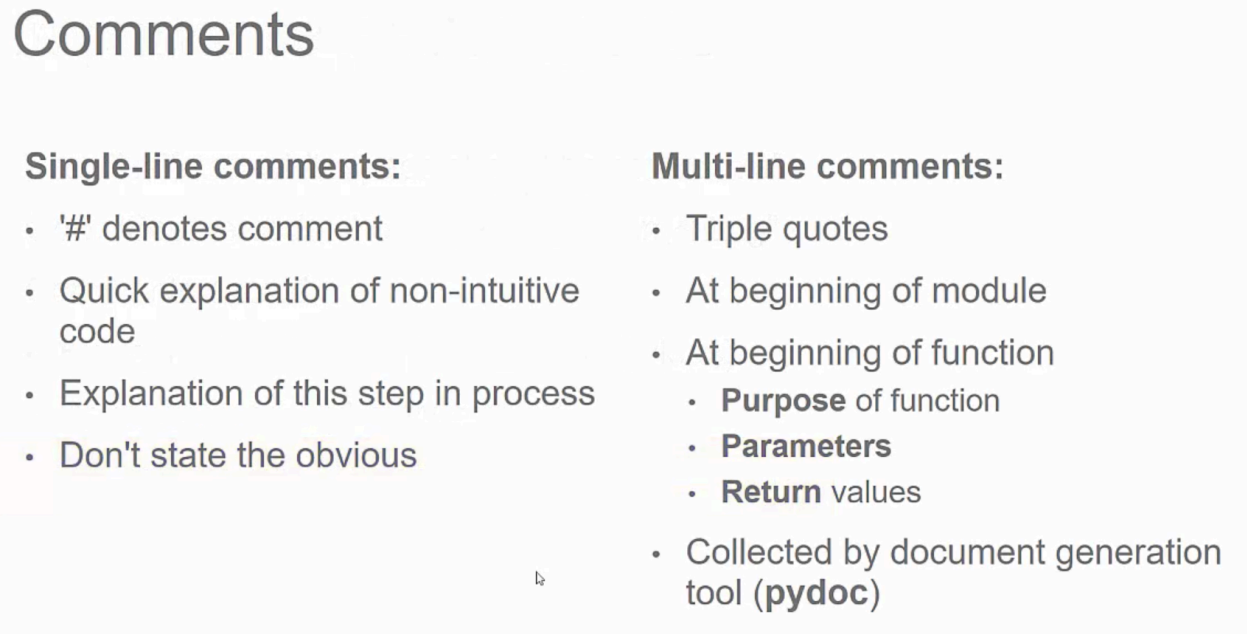


Importance of Comments

Programmers have differing opinions regarding comments: their usefulness, their reliability, their value to novice and experienced developers. Some would say that the code itself should be self-documenting; and some would say that comments should constitute most of non-whitespace characters in an application. Most programmers are somewhere in between.

But there are a few aspects of commenting code that most programmers can agree on:

- **Explaining:** Comments should be used to explain the purpose of a line or section of code. Comments such as **x = y # set x equal to y** are not very useful. routes = {} # create dictionary to hold routing info Has more value to the person hoping to understand the code.
- **Maintenance:** Comments should be maintained, meaning that if the code changes, make sure the comments related to the code change as well. An over-abundance of unnecessary comments makes maintenance overly difficult.
- **Auto-generation:** Most programming languages have some type of automatic documentation generation. Java has Javadocs, and Python has Pydoc. Create your comments in such a way as to facilitate this capability.



Comments

In Python, comments can come in two forms: single-line and multi-line. In brief:

Single-Line Comments

- Denoted by '#'; everything following, up to the end of the line, is considered part of the comment.
- Useful for quick explanation of the purpose of a piece of code if it is not obvious.
- Useful for describing a step-by-step process, inline with the code, to facilitate understanding of the overall sequence of what is going on.
- Comments stating the obvious are of little value, and actually may be detrimental, because of the overhead of maintaining such things.

Multi-line Comments

- Use triple-quote marks to identify the beginning and end of a multi-line comment.
- Useful for module and function descriptions at the beginning.
- For modules, include the purpose of the module, and what it contains.
- For functions, include the purpose of the function, and also the parameters that are passed, and the values returned, by that function.
- Multi-line comments will be gathered by the 'pydoc' tool when run against your application or code, and will generate a nicely formatted, easy-to-read piece of documentation regarding your module or function.

But be aware that that documentation will only be as good as what you put there. Meaningless multi-line comments will result in nicely formatted, meaningless documentation.

Here is an example of well-documented code , with single and multi-line comments:

```
#!/usr/bin/env python

"""
Example of a script that executes a CLI command on a remote
device over established SSH connection.

Administrator login options and CLI commands are device specific,
thus this script needs to be adapted to a concrete device specifics.
Current script assumes interaction with Cisco IOS device.

NOTES: Requires installation of the 'paramiko' Python package:
pip install paramiko
       The 'paramiko' package is documented at:
http://docs.paramiko.org
       Complete set of SSH client operations is available at:
http://docs.paramiko.org/en/1.15/api/client.html

command_ssh.py
"""
```

```
# built-in modules
import time
import socket

# third-party modules
import paramiko

def enable_privileged_commands(device_info, rsh):

    """Turn on privileged commands execution.
    :param dict device_info: dictionary containing information
        about target device.
    :param paramiko.channel.Channel rsh: channel connected to a remote shell.
    """

    cmd = "enable\n"
    # Execute the command (wait for command to complete)
    rsh.send(cmd)
    time.sleep(1)
    output = rsh.recv(device_info['max_bytes'])
    if(device_info['password_prompt'] in output):
        password = "%s\n" % device_info['password']
        rsh.send(password)
        rsh.recv(device_info['max_bytes'])
```