Part 47
-
Python Notes

Version: 2020-02-29

My Python Working Notes

PIP
PIP is a package manger used to install Python modules and packages.
PIP normally comes installed with Python. If it isn't already installed you can install it by following these instructions.
`pip` is already installed if you are using Python 2.7.9 or higher or Python 3.4 or higher, downloaded from python.org or if you are working in a Virtual Environment created by virtualenv or pyvenv. Just make sure to upgrade pip.

First make sure distutils are installed.

```
apt -y install python-distutils
apt -y install python3-distutils
```

To install pip, securely download `get-pip.py` from

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

Then run the following:

```
python get-pip.py
```

Warning: Be cautious if you are using a Python install that is managed by your operating system or another package manager. `get-pip.py` does not coordinate with those tools, and may leave your system in an inconsistent state.

`get-pip.py` also installs `setuptools` and `wheel` if they are not already. `setuptools` is required to install source distributions. Both are required in order to build a `Wheel Cache` (which improves installation speed), although neither are required to install pre-built `wheels`.

Note**:** The `get-pip.py` script is supported on the same python version as `pip`. For the now unsupported Python 2.6, alternate script is available from
`https://bootstrap.pypa.io/2.6/get-pip.py`

get-pip.py options

```
--no-setuptools
If set, do not attempt to install setuptools

--no-wheel
If set, do not attempt to install wheel
```

`get-pip.py` allows pip install options and the general options. Below are some examples:

Install from local copies of pip and setuptools:

```
python get-pip.py --no-index --find-links=/local/copies
```

Install to the user site:

```
python get-pip.py --user
```

Install behind a proxy:

```
python get-pip.py –proxy="http://[user:passwd@]proxy.server:port"
```

`get-pip.py` can also be used to install a specified combination of `pip`, `setuptools`, and `wheel` using the same requirements syntax as `pip`:

```
python get-pip.py pip==9.0.2 wheel==0.30.0 setuptools==28.8.0
```

Upgrading pip

```
pip install -U pip
```

Multiple Python Versions and Installs

You can have multiple python versions installed on the same machine, but it can cause problems if you aren't very experienced.
One of the common problems I encountered was when I installed a package with PIP it would be installed to the wrong version/location.
You can use the command

```
pip –version
```

To see where pip will install new packages

```
pip 19.3.1 from /usr/local/lib/python3.7/dist-packages/pip (python 3.7)
```

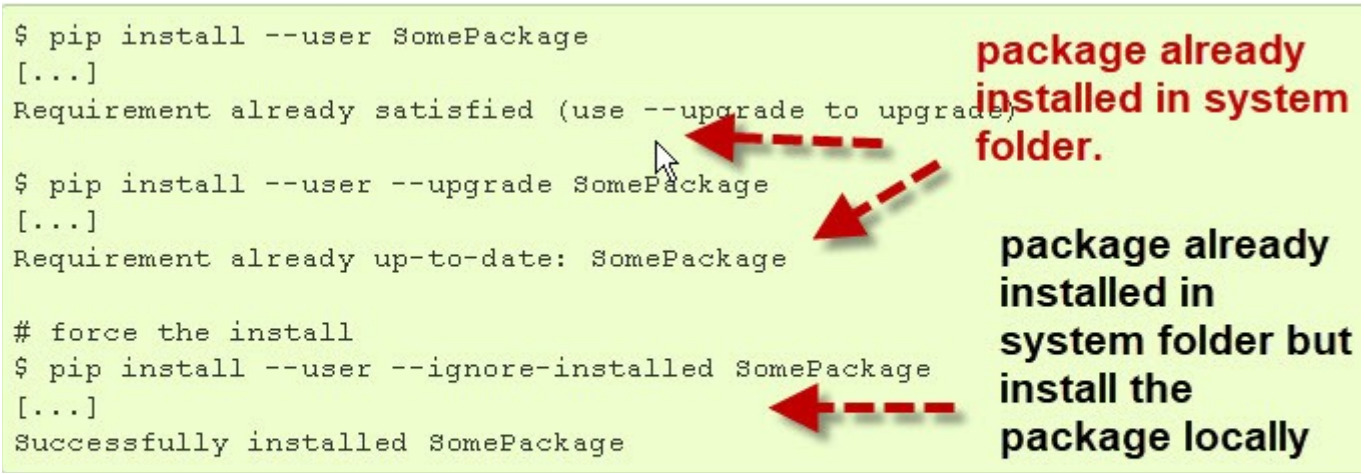
So pip will install new packages in `/usr/local/lib/python3.7/dist-packages/pip`

To see where pip has installed a package type:

```
pip show package_name
```

PIP on Linux

On linux packages can be installed globally and locally.
Global packages are available to all users whereas local packages are restricted to the user that installed them.
Doting a simple `pip install` will usually try to install the package in a system folder usually `usr/local/lib/python3.7/` which requires root access.
You can run as root using `sudo` to gain permissions to make the install work.
However another option and a better one is to install the package locally for that user and to do that you can use the `--user` option to tell `pip` that it is a local install.
The following screen shot taken from the PIP documentation shows some scenarios when you try to install a package using the `--user` option that is already available in a system folder.
The default action for `pip` is not to do the install as it doesn't need to.



PIP versions and install Location
You will find different versions of pip installed on your system usually pip, pip3, pip3.7.
Although you will see instructions saying do

```
pip install packagename
```

Following those instructions may not work because pip will install the package under the wrong version of python.
So doing

```
pip install paho-mqtt
```

May install the paho-mqtt module under python2.7, python3.7 depending on the versions you have on the system and your location when running the command.

Alternatively use pip3.7 if available. Pip3.7 is simply a shortcut or link to the correct pip.
Again using the command `pip –version` will show you where pip will install the files

Common pip commands

```
pip install package-name
pip uninstall package-name
pip install -U pip                  #upgrade pip
```

On Linux if you run as `sudo` and `pip3` then the packages are installed in the `/usr/local/lib/python3.7/dist-packages` folder.
In this case python3.7 is the latest version.
If you don't run as `sudo` then they installed in the home folder under `.local/lib/python3.7/dist-packages`.

`So`, when using `pip` be aware that it is important to check where pip is installing your packages.

Linux Folders and Python

`usr/bin:`	has binaries and links from generic e.g 2 to 2.7 etc
`usr/lib:`	main python files and also the idle
`/usr/local/lib/:`	distribution packages that you have installed. E.g the paho mqtt client would be installed there.
`.local/lib`	locally installed packages (–user option)

Python Packages

Packages
A python package is a collection of python modules. It is effectively a folder containing python files.
A python package directory has to have a **`__init__.py`** file.
The presence of this file indicates to the Python interpreter that the directory/folder contains python modules.
Although the `__init__.py` file can contain configuration information is is also often completely blank.
The `__init__.py` file makes Python treat directories containing it as modules. Furthermore, this is the first file to be loaded in a module, so you can use it to execute code that you want to run each time a module is loaded, or specify the submodules to be exported

Modules
A python module is a file containing python code. A module can define classes, functions and variables.
A module can be imported into another Python program or run directly.

Namespaces and Modules
A module also defines a **namespace**. Two modules can contain functions and variable with the same name as they are effectively in two separate namespaces.
Example two modules, **`m1`** **and** **`m2`** each contain a function **`add()`**.
To access the add function from modules **m1** you need to use the following syntax:
`m1.add()`
To access the add function from modules **m2** you need to use the following syntax:
`m2.add()`
You might find this useful: `https://www.tutorialspoint.com/python/python_modules.htm`

Importing Modules
When using the **`import module`** format then the module is imported with its namespace and so to access a function in that module you will need to prefix it with the module name.

If you use the **`from module import function`** format then you can access the function **without** prefixing it with the module name.

The __main__ module
When you run a script by typing:

```
python myscript.py
```

The Python interpreter runs it as module called `__main__` which gets its own namespace.
If you imported the script then it would have the module name `myscript`.
You will also encounter the following code:

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

This is found in python modules that can be imported or executed directly.

Where are my packages being installed?
On linux the packages will be installed either in one of these locations:

```
usr/local/lib/python3.7/dist-packages
usr/lib/python3/dist-packages
usr/lib/python3.7/dist-packages
```

You can find out where Python expects the packages to be found using the following simple script

```
import site
print (site.getsitepackages())
```

You can also do it from the python command prompt as follows:

```
root@Debian10:~# python
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
import site
site.getsitepackages()
['/usr/local/lib/python3.7/dist-packages', '/usr/lib/python3/dist-packages',
'/usr/lib/python3.7/dist-packages']
```

Running Scripts
From the command line on Windows or linux use:

```
python myprog.py
```

If you have several versions on Python available e.g. python 2 and 3, then you can use

```
python3 myprog.py
```

to start your program using the Python3 Interpreter.

To find out which version of python is the default version use:

```
python –version
```

If your script needs a particular version to work then use:

```
import sys
assert sys_version_info >=(2,5)
```

The SheBang Line
At the top of scripts written for Linux you often see this line.

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

This lets you run your program without starting the interpreter so you use simply:

```
myprog.py
```

However the above may not work if you have multiple python versions installed as it will use the default version. In this case you will need to be explicit and use

```
/usr/bin/python3.7
```

to run using python3.7

A useful addition to you scripts is the following lines which will print out the version of python the script is using:

```
import sys
print(sys.executable)
```

Exiting Python Scripts
The best seems to be:

```
raise SystemExit
```

Let me give some information on them:

1. `quit()` raises the `SystemExit` exception behind the scenes.
   Furthermore, if you print it, it will give a message:

   ```
   print (quit)
   Use quit() or Ctrl-Z plus Return to exit
   ```

   This functionality was included to help people who do not know Python. After all, one of the most likely things a newbie will try to exit Python is typing in quit.
   Nevertheless, quit should not be used in production code. This is because it only works if the site module is loaded. Instead, this function should only be used in the interpreter.

2. `exit()` is an alias for `quit` (or vice-versa). They exist together simply to make Python more user-friendly.
   Furthermore, it too gives a message when printed:

   ```
   print (exit)
   Use exit() or Ctrl-Z plus Return to exit
   ```

   However, like quit, exit is considered bad to use in production code and should be reserved for use in the interpreter. This is because it too relies on the site module.

3. `sys.exit()` raises the `SystemExit` exception in the background. This means that it is the same as quit and exit in that respect.
   Unlike those two however, sys.exit is considered good to use in production code. This is because the sys module will always be there.

4. `os._exit()` exits the program without calling cleanup handlers, flushing stdio buffers, etc. Thus, it is not a standard way to exit and should only be used in special cases. The most common of these is in the child process(es) created by `os.fork`.
   Note that, of the four methods given, only this one is unique in what it does.

All of these can be called without arguments, or you can specify the exit status, e.g., exit(1) or raise SystemExit(1) to exit with status 1. Note that portable programs are limited to exit status codes in the range 0-255, if you raise SystemExit(256) on many systems this will get truncated and your process will actually exit with status 0.

Summed up, all four methods exit the program. However, the first two are considered bad to use in production code and the last is a non-standard, dirty way that is only used in special scenarios. So, if you want to exit a program normally, go with the third method: sys.exit.
Or, even better in my opinion, you can just do directly what sys.exit does behind the scenes and run:

```
raise SystemExit
```

This way, you do not need to import sys first.
However, this choice is simply one on style and is purely up to you.

The Pythonic Guide To Logging
When done properly, logs are a valuable component of your observability suite. Logs tell the story of how data has changed in your application. They let you answer questions such as: Why did John Doe receive an error during checkout yesterday? What inputs did he provide? The practice of logging ranges from very simple static string statements to rich structured data events. We'll explore how logging in Python will give you a better view of what's going on in your application. In addition, I'll explore some best practices that will help you get the most value out of your logs using the built in Python module.

Print('Why Not Just Use The Print Statement?')
Many Python tutorials show readers how to use print as a debugging tool. Here's an example using print print an exception as a script:
import sys

```
def captureException():
...     return sys.exc_info()
...
try:
...     1/0
except:
...     print(captureException())
...
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',),
<traceback object at 0x101acb1c8>)
None
```

While this is useful in smaller scripts, as your application and operation requirements grow, print becomes a less viable solution. It doesn't offer you the flexibility to turn off entire categories of output statements and it only allows you to output to the stdout. It also misses information such as line number and time that it was generated that can assist with debugging. While print is the easiest approach since it doesn't require setup, it can quickly come back to bite you. It's also bad practice to ship a package that prints directly to stdout because it removes the user's ability to control the messages.

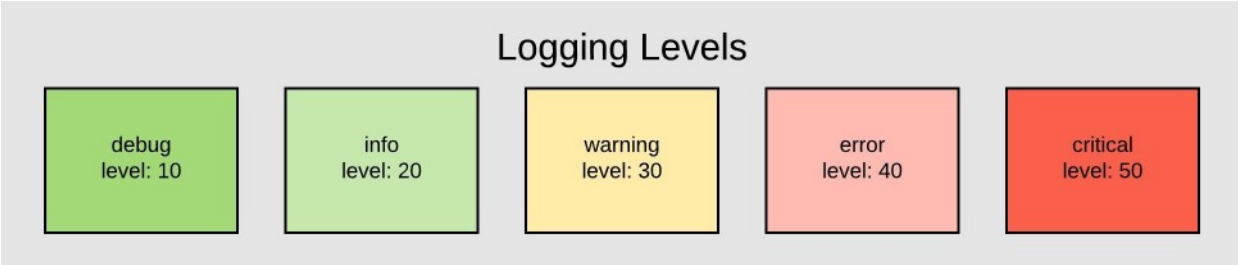
Logging.Info("Hello World To Logging")
The logging module is part of Python's standard library and has been since version 2.3. It automatically adds context, such as line number & timestamps, to your logs. The module makes it easy to add namespace logging and severity levels to give you more control over where and what is output.
I'm a believer that the best way to learn something is to do it, so I'd encourage you to follow along in the Python REPL. Getting started with the logging module is simple, here's all you have to do:

```
import logging
logging.basicConfig()
logger = logging.getLogger(__name__)

logger.critical('logging is easier than I was expecting')
CRITICAL:__main__:logging is easier than I was expecting
```

What just happened? getLogger provided us with a logger instance. We then gave it the event 'logging is easier than I was expecting' with a level of critical.

Levels
The Python module allows you to differentiate events based on their severity level. The levels are represented as integers between 0 and 50. The module defines five constants throughout the spectrum, making it easy to differentiate between messages.



Each level has meaning attached to it, and you should think critically about the level you're logging at.

```
logger.critical("this better be bad")
CRITICAL:root:this better be bad

logger.error("more serious problem")
ERROR:root:more serious problem

logger.warning("an unexpected event")
WARNING:root:an unexpected event

logger.info("show user flow through program")
logger.debug("used to track variables when coding")
```

Notice the info and debug didn't print a message. By default, the logger will only print warning, error, or critical messages. You can customize this behavior, and even modify it at runtime to activate more verbose logging dynamically.

```
# should show up; info is a higher level than debug
logger.setLevel(logging.DEBUG)
logger.info(1)
INFO:root:1

# shouldn't show up; info is a lower level than warning
... logger.setLevel(logging.WARNING)
logger.info(2)
```

Formatting The Logs
The default formatter is not useful for formatting your logs, because it doesn't include critical information. The logging module makes it easy for you to add it in by changing the format. We'll set the format to show the time, level, and message:

```
import logging
logFormatter = '%(asctime)s - %(levelname)s - %(message)s'
logging.basicConfig(format=logFormatter, level=logging.DEBUG)
logger = logging.getLogger(__name__)
logger.info("test")
2018-06-19 17:42:38,134 - INFO - test
```

Some of this data, such as time and levelname can be captured automatically, but you can (and should) push extra context to your logs.

Adding Context
A generic log message provides just about as little information as no log message. Imagine if you had to go through your logs and you saw removed from cart. This makes it difficult to answer questions such as: *When was the item removed? Who removed it? What did they remove?*
It's best to add structured data to your logs instead of string-ifying an object to enrich it. Without structured data, it's difficult to decipher the log stream in the future. The best way to deal with this is to push important metadata to the extra object. Using this, you'll be able to enrich your messages in the stream.

```
import logging
logFormatter = '%(asctime)s - %(user)s - %(levelname)s - %(message)s'
logging.basicConfig(format=logFormatter, level=logging.DEBUG)
logger = logging.getLogger(__name__)
logger.info('purchase completed', extra={'user': 'Sid Panjwani'})
2018-06-19 17:44:10,276 - Sid Panjwani - INFO - purchase completed
```

Performance
Logging introduces overhead that needs to be balanced with the performance requirements of the software you write. While the overhead is generally negligible, bad practices and mistakes can lead to unfortunate situations. Here are a few helpful tips:

Wrap Expensive Calls in a Level Check
The Python Logging Documentation recommends that expensive calls be wrapped in a level check to delay evaluation.

```
if logger.isEnabledFor(logging.INFO):
    logger.debug('%s', expensive_func())
```

Now, expensive_func will only be called if the logging level is greater than or equal to INFO.

Avoid Logging in the Hot Path
The hot path is code that is critical for performance, so it is executed often. It's best to avoid logging here because it can become an IO bottleneck, unless it's necessary because the data to log is not available outside the hot path.

Storing & Accessing These Logs
Now that you've learned to write these (beautiful) logs, you've got to determine what to do with them. By default, logs are written to the standard output device (probably your terminal window), but Python's logging module provides a rich set of options to customize the output handling. Logging to standard output is encouraged, and platforms such as Heroku, Amazon Elastic Beanstalk, and Docker build on this standard by capturing the standard output and redirecting to other logging facilities at a platform level.

Logging To A File
The logging module makes it easy to write your logs to a local file using a "handler" for long-term retention.

```
import logging
logger = logging.getLogger(__name__)

handler = logging.FileHandler('myLogs.log')
handler.setLevel(logging.INFO)

logger.addHandler(handler)
logger.info('You can find this written in myLogs.log')
```

If you've been following along (which you should be), you'll notice that your log won't show up in your file because it is level info. Make sure to use setLevel to change that. Now it's easy to search through your log file using grep.

```
grep critical myLogs.log
```

Now you can search for messages that contain keywords such as critical or warning.

Rotating Logs

The Python logging module makes it easy to log in a different file after an interval of time or after the log file reaches a certain size. This becomes useful if you automatically want to get rid of older logs, or if you're going to search through your logs by date since you won't have to search through a huge file to find a set of logs that are already grouped.

To create a handler that makes a new log file every day and automatically deletes logs more than five days old, you can use a TimedRotatingFileHandler. Here's an example:

```
logger = logging.getLogger('Rotating Log by Day')

# writes to pathToLog
# creates a new file every day because `when="d"` and `interval=1`
# automatically deletes logs more than 5 days old because
#`backupCount=5`
handler = TimedRotatingFileHandler(pathToLog, when="d", interval=1,
backupCount=5)
```

The Drawbacks

It's important to understand where logs fit in when it comes to observing your application. Logs are a component of your observability stack, but metrics and tracing are equally so. Logging can become expensive, especially at scale. Metrics can be used to aggregate data and answer questions about your customers without having to keep a trace of every action, while tracing enables you to see the path of your request throughout your platform.

Wrapping Up

If you take anything away from this post, it should be that logs serve as the *source of truth* for the user's actions. Even for ephemeral actions, such as putting an item into and out of a cart, it's essential to be able to trace the user's steps for a bug report, and the logs allow you to determine their actions.

The Python logging module makes this easy. It allows you to format your logs, dynamically differentiate between messages using levels, and ship your logs externally using "handlers". Though not the only mechanism you should be using to gain insight into user actions, it is an effective way to capture raw event data and answer unknown questions.

Some Python Best Practices, Tips, And Tricks

https://towardsdatascience.com/30-python-best-practices-tips-and-tricks-caefb9f8c5f5

Check for a minimum required Python version
You can check for the Python version in your code, to make sure your users are not running your script with an incompatible version. Use this simple check:

```
if not sys.version_info > (2, 7):
   # berate your user for running a 10 year python version
elif not sys.version_info >= (3, 5):
   # Kindly tell your user (s)he needs to upgrade
   # because you're using 3.5 features
```

List Comprehensions
A list comprehension can replace ugly for loops used to fill a list. The basic syntax for a list comprehension is:

```
[ expression for item in list if conditional ]
```

A very basic example to fill a list with a sequence of numbers:

```
mylist = [i for i in range(10)]
print(mylist)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

And because you can use an expression, you can also do some math:

```
squares = [x**2 for x in range(10)]
print(squares)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Or even call an external function:

```
def some_function(a):
    return (a + 5) / 2

my_formula = [some_function(i) for i in range(10)]
print(my_formula)
# [2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0]
```

And finally, you can use the ‘if’ to filter the list. In this case, we only keep the values that are dividable by 2:

```
filtered = [i for i in range(20) if i%2==0]
print(filtered)
# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Check memory usage of your objects
With `sys.getsizeof()` you can check the memory usage of an object:
Woah… wait… why is this huge list only 48 bytes?
It's because the range function returns a *class* that only behaves like a list. A range is a lot more memory efficient than using an actual list of numbers.
You can see for yourself by using a list comprehension to create an actual list of numbers from the same range:

```
import sys

myreallist = [x for x in range(0, 10000)]
print(sys.getsizeof(myreallist))
# 87632
```

Return multiple values
Functions in Python can return more than one variable without the need for a dictionary, a list or a class. It works like this:

```
def get_user(id):
    # fetch user from database
    # ....
    return name, birthdate

name, birthdate = get_user(4)
```

This is alright for a limited number of return values. But anything past 3 values should be put into a (data) class.

Use data classes
Since version 3.7, Python offers data classes. There are several advantages over regular classes or other alternatives like returning multiple values or dictionaries:

- a data class requires a minimal amount of code
- you can compare data classes because __eq__ is implemented for you
- you can easily print a data class for debugging because __repr__ is implemented as well
- data classes require type hints, reduced the chances of bugs

Here's an example of a data class at work:

```
from dataclasses import dataclass

@dataclass
class Card:
    rank: str
    suit: str

card = Card("Q", "hearts")

print(card == card)
# True

print(card.rank)
# 'Q'

print(card)
Card(rank='Q', suit='hearts')
```

In place variable swapping
A neat little trick that can save a few lines of code:

```
a = 1
b = 2
a, b = b, a
print (a)
# 2
print (b)
# 1
```

Merging dictionaries (*Python 3.5+)*
Since Python 3.5, it became easier to merge dictionaries:

```
dict1 = { 'a': 1, 'b': 2 }
dict2 = { 'b': 3, 'c': 4 }
merged = { **dict1, **dict2 }
print (merged)
# {'a': 1, 'b': 3, 'c': 4}
```

If there are overlapping keys, the keys from the first dictionary will be overwritten.

String to title case
This is just one of those lovely gems:

```
mystring = "10 awesome python tricks"
print(mystring.title())
'10 Awesome Python Tricks'
```

Split a string into a list
You can split a string into a list of strings. In this case, we split on the *space* character:

```
mystring = "The quick brown fox"
mylist = mystring.split(' ')
print(mylist)
# ['The', 'quick', 'brown', 'fox']
```

To split on whitespace, you actually don't have to give split any arguments. By default, all runs of consecutive whitespace are regarded as a single whitespace separator by split. So we could just as well use `mystring.split()`.

Create a string from a list of strings
And vice versa from the previous trick, create a string from a list and put a space character between each word:

```
mylist = ['The', 'quick', 'brown', 'fox']
mystring = " ".join(mylist)
print(mystring)
# 'The quick brown fox'
```

If you were wondering why it's not `mylist.join(" ")` — good question!
It comes down to the fact that the String.join() function can join not just lists, but any iterable. Putting it inside String prevents implementing the same functionality in multiple places.

Slicing a list
The basic syntax of list slicing is:

```
a[start:stop:step]
```

Start, stop and step are optional. If you don't fill them in, they will default to:

- 0 for start
- the end of the string for end
- 1 for step

```
Here are some examples:
# We can easily create a new list from
# the first two elements of a list:
first_two = [1, 2, 3, 4, 5][0:2]
print(first_two)
# [1, 2]

# And if we use a step value of 2,
# we can skip over every second number
# like this:
steps = [1, 2, 3, 4, 5][0:5:2]
print(steps)
# [1, 3, 5]

# This works on strings too. In Python,
# you can treat a string like a list of
# letters:
mystring = "abcdefdn nimt"[::2]
print(mystring)
# 'aced it'
```

Reversing strings and lists
You can use the slice notation from above to reverse a string or list. By using a negative stepping value of -1, the elements are reversed:

```
revstring = "abcdefg"[::-1]
print(revstring)
# 'gfedcba'

revarray = [1, 2, 3, 4, 5][::-1]
print(revarray)
# [5, 4, 3, 2, 1]
```

Using map()
One of Python's built-in functions is called `map()`. The syntax for `map()` is:

```
map(function, something_iterable)
```

So you give it a function to execute, and something to execute on. This can be anything that's iterable. In the examples below I'll use a list.
Take a look at your own code and see if you can use `map()` instead of a loop somewhere!

```
def upper(s):
    return s.upper()

mylist = list(map(upper, ['sentence', 'fragment']))
print(mylist)
# ['SENTENCE', 'FRAGMENT']

# Convert a string representation of
# a number into a list of ints.
list_of_ints = list(map(int, "1234567")))
print(list_of_ints)
# [1, 2, 3, 4, 5, 6, 7]
```

Get unique elements from a list or string
By creating a set with the set() function, you get all the unique elements from a list or list-like object:

```
mylist = [1, 1, 2, 3, 4, 5, 5, 5, 6, 6]
print (set(mylist))
# {1, 2, 3, 4, 5, 6}

# And since a string can be treated like a
# list of letters, you can also get the
# unique letters from a string this way:
print (set("aaabbbcccdddeeefff"))
# {'a', 'b', 'c', 'd', 'e', 'f'}
```

Find the most frequently occurring value

To find the most frequently occurring value in a list or string:

```
test = [1, 2, 3, 4, 2, 2, 3, 1, 4, 4, 4]
print(max(set(test), key = test.count))
# 4
```

Do you understand why this works? Try to figure it out for yourself before reading on.
You didn't try, did you? I'll tell you anyway:

- `max()` will return the highest value in a list. The key argument takes a single argument function to customize the sort order, in this case, it's `test.count`. *The function is applied to each item on the iterable.*
- `test.count` is a built-in function of list. It takes an argument and will count the number of occurrences for that argument. So `test.count(1)` will return 2 and `test.count(4)` returns 4.
- `set(test)` returns all the unique values from test, so {1, 2, 3, 4}

So what we do in this single line of code is take all the unique values of test, which is `{1, 2, 3, 4}`. Next, max will apply the list.count function to them and return the maximum value.

Create a progress bar
You can create your own progress bar, which is fun to do. But it's quicker to use the *progress* package:

```
pip3 install progress
```

Now you can create a progress bar with minimal effort:
The following animation demonstrates all the available progress types:

```
from progress.bar import Bar

bar = Bar('Processing', max=20)
for i in range(20):
    # Do some work
    bar.next()
bar.finish()
```

Multi-Line Strings
Although you can use triple quotes to include multi-line strings in your code, it's not ideal. Everything you put between the triple quotes becomes the string, including the formatting, as you can see below.
I prefer the second way, which concatenates multiple lines together, allowing you to format your code nicely. The only downside is that you need to explicitly put in newlines.

```
s1 = """Multi line strings can be put
        between triple quotes. It's not ideal
        when formatting your code though"""

print (s1)
# Multi line strings can be put
#         between triple quotes. It's not ideal
#         when formatting your code though

s2 = ("You can also concatenate multiple\n" +
        "strings this way, but you'll have to\n"
        "explicitly put in the newlines")

print(s2)
# You can also concatenate multiple
# strings this way, but you'll have to
# explicitly put in the newlines
```

Ternary Operator For Conditional Assignment
This is another one of those ways to make your code more concise while still keeping it readable:

```
[on_true] if [expression] else [on_false]
```

As an example:

```
x = "Success!" if (y == 2) else "Failed!"
```

Counting occurrences
You can use `Counter` from the collections library to get a dictionary with counts of all the unique elements in a list:

```
from collections import Counter

mylist = [1, 1, 2, 3, 4, 5, 5, 5, 6, 6]
c = Counter(mylist)
print(c)
# Counter({1: 2, 2: 1, 3: 1, 4: 1, 5: 3, 6: 2})

# And it works on strings too:
print(Counter("aaaaabbbbbccccc"))
# Counter({'a': 5, 'b': 5, 'c': 5})
```

Chaining of comparison operators
You can chain comparison operators in Python, creating more readable and concise code:

```
x = 10

# Instead of:
if x > 5 and x < 15:
    print("Yes")
# yes

# You can also write:
if 5 < x < 15:
    print("Yes")
# Yes
```

Working with dates
The `python-dateutil` module provides powerful extensions to the standard datetime module. Install it with:

```
pip3 install python-dateutil
```

You can do so much cool stuff with this library. I'll limit the examples to just this one that I found particularly useful: fuzzy parsing of dates from log files and such.
Just remember: where the regular Python datetime functionality ends, python-dateutil comes in!

```
from dateutil.parser import parse

logline = 'INFO 2020-01-01T00:00:01 Happy new year, human.'
timestamp = parse(log_line, fuzzy=True)
print(timestamp)
# 2020-01-01 00:00:01
```

Integer division
In Python 2, the division operator (/) defaults to an integer division, unless one of the operands is a floating-point number. So you have this behavior:

```
# Python 2
5 / 2 = 2
5 / 2.0 = 2.5
```

In Python 3, the division operator defaults to a floating-point division and the // operator has become an integer division. So we get:

```
# Python 3
5 / 2 = 2.5
5 // 2 = 2
```

Charset detection with chardet
You can use the chardet module to detect the charset of a file. This can come in very useful when analyzing big piles of random text. Install with:

```
pip install chardet
```

You now have an extra command-line tool called chardetect, which can be used like this:

```
chardetect somefile.txt
somefile.txt: ascii with confidence 1.0
```