

# INTRODUCTION TO PYTHON PROGRAMMING LANGUAGE

Most of the slides are from  
Caitlin Carnahan, Florida  
State University

# LOGGING AND UNIT TEST

## LOGGING

Logging is an essential practice for non-trivial applications in which events are recorded for potential diagnostic use in the future.

Logging can be used to record the following kinds of items:

- Errors: any exceptions that are raised can be logged when they are caught.
- Significant events: for example, when an administrator logs into a system.
- Data Handled: for example, a request came into the system with x, y, z parameters.

Logging can provide useful information about the state of the program during a crash or help a developer understand why the program is exhibiting some kind of behavior.

## LOGGING

The Python Standard Library actually comes with a standard logging module. It is a particularly good decision to use this module because it can include messages generated from any other package that also uses this library.

As usual, you can use the logging module by using the import logging statement.

## LOGGING

Here's a simple example of some logging calls.

```
>>> import logging
>>> logging.critical("Imminent fatal error!")
CRITICAL:root:Imminent fatal error!
>>> logging.error("Could not perform some function but still running.")
ERROR:root:Could not perform some function but still running.
>>> logging.warning("Disk space low...")
WARNING:root:Disk space low...
```

## LOGGING

There are five logging levels which should be used accordingly to reflect the severity of the event being logged.

Level	Use
Debug	For development.
Info	Messages confirming expected behavior.
Warning	Warnings about future issues or benign unexpected behavior.
Error	Something unexpected happened and software is not working properly as a result.
Critical	Something unexpected happened and software is not running anymore as a result.

The `logging.loglevel()` methods are the easiest ways to quickly log a message with a given severity level. You can also use `logging.log(level, msg)`.

## LOGGING

Note that the info and debug messages are not logged. This is because the default logging level – the level at which messages must be logged – is warning and higher.

```
>>> import logging
>>> logging.critical("Imminent fatal error!")
CRITICAL:root:Imminent fatal error!
>>> logging.error("Could not perform some function but still running.")
ERROR:root:Could not perform some function but still running.
>>> logging.warning("Disk space low...")
WARNING:root:Disk space low...
>>> logging.info("Everything seems to be working ok.")
>>> logging.debug("Here's some info that might be useful to debug with.")
```

## LOGGING

It is not typical to log directly in standard output. That might be terrifying to your client. You should at least direct logging statements to a file. Take the module `logtest.py` for example:

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

## LOGGING

It is not typical to log directly in standard output. That might be terrifying to your client. You should at least direct logging statements to a file. Take the module `logtest.py` for example:

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

Log debug  
messages and  
higher

## LOGGING

After running `logtest.py`, we have a logfile called `example.log` with the following contents:

```
CRITICAL:root:Imminent fatal error!
ERROR:root:Could not perform some function but still running.
WARNING:root:Disk space low...
INFO:root:Everything seems to be working ok.
DEBUG:root:Here's some info that might be useful to debug with.
```

## LOGGING

```
import logging
import sys

for arg in sys.argv:
    if arg[:11] == "--loglevel=":
        loglvl = arg[11:].upper()
    else:
        loglvl = "INFO"

logging.basicConfig(filename='example.log', level=getattr(logging, loglvl))
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

An even better approach would be to allow the loglevel to be set as an argument to the program itself.

## LOGGING

Now, I can specify what the logging level should be without changing the code – this is a more desirable scenario. If I do not set the logging level, it will default to INFO.

```
~$ python logtest.py --loglevel=warning
~$ more example.log
CRITICAL:root:Imminent fatal error!
ERROR:root:Could not perform some function but still running.
WARNING:root:Disk space low...
```

## LOGGING

The `logging.basicConfig()` function is the simplest way to do basic global configuration for the logging system.

Any calls to `info()`, `debug()`, etc will call `basicConfig()` if it has not been called already. Any subsequent calls to `basicConfig()` have no effect.

Argument	Effect
filename	File to which logged messages are written.
filemode	Mode to open the filename with. Defaults to 'a' (append).
format	Format string for the messages.
datefmt	Format string for the timestamps.
level	Log level messages and higher.
stream	For StreamHandler objects.

## LOGGING

If your application contains multiple modules, you can still share a single log file. Let's say we have a module `driver.py` which uses another module `mymath.py`.

```
import logging
import mymath

def main():
    logging.basicConfig(filename='example.log', level=logging.DEBUG)
    logging.info("Starting.")
    x = mymath.add(2, 3)
    logging.info("Finished with result " + str(x))

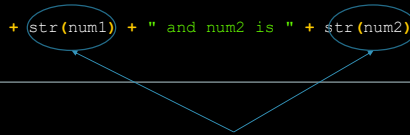
if __name__ == "__main__":
    main()
```

## LOGGING

The mymath.py module also logs some message but note that we do not have to reconfigure the logging module. All the messages will log to the same file.

```
import logging

def add(num1, num2):
    logging.debug("num1 is " + str(num1) + " and num2 is " + str(num2))
    logging.info("Adding.")
    return num1 + num2
```



Note the logging of variable data here.

## LOGGING

This behavior gives us some insight into the reason why additional calls `basicConfig()` have no effect. The first `basicConfig()` call made during the execution of the application is used to direct logging for all modules involved – even if they have their own `basicConfig()` calls.

```
~$ python driver.py
~$ more example.log
INFO:root:Starting.
DEBUG:root:num1 is 2 and num2 is 3
INFO:root:Adding.
INFO:root:Finished with result 5
```



## LOGGING

You can modify the format of the message string as well. Typically, it's useful to include the level, timestamp, and message content.

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG,
                    format='%(asctime)s: %(levelname)s: %(message)s')
logging.info("Some important event just happened.")
```

```
~? python logtest.py
~? more example.log
2015-6-11 11:41:42,612:INFO:Some important event just happened.
```

## LOGGING

All of the various formatting options can be found [here](#).

This is really just a very basic usage of the logging module, but its definitely enough to log a small project.

Advanced logging features give you a lot more control over when and how things are logged – most notably, you could implement a rotating series of log files rather than one very large logfile which might be difficult to search through.

## AUTOMATED TESTING

Obviously, after you write some code, you need to make sure it works. There are pretty much three ways to do this, as pointed out by Ned Batchelder:

- Automatically test your code.
- Manually test your code.
- Just ship it and wait for clients to complain about your code.

The last is...just not a good idea. The second can be downright infeasible for a large project. That leaves us with automated testing.

## TESTING

Let's say we have the following module with two simple functions.

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

## TESTING

The simplest way to test is to simply pop open the interpreter and try it out.

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False

>>> import even
>>> even.even(2)
True
>>> even.even(3)
False
>>> even.pos_even(2)
True
>>> even.pos_even(3)
False
>>> even.pos_even(-2)
False
>>> even.pos_even(-3)
False
```

## TESTING

This method is time-consuming and not repeatable. We'll have to redo these steps manually anytime we make changes to the code.

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False

>>> import even
>>> even.even(2)
True
>>> even.even(3)
False
>>> even.pos_even(2)
True
>>> even.pos_even(3)
False
>>> even.pos_even(-2)
False
>>> even.pos_even(-3)
False
```

## TESTING

We can store the testing statements inside of a module and run them anytime we want to test. But this requires us to manually "check" the correctness of the results.

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

test\_even.py

```
import even
print "even.even(2) = ", even.even(2)
print "even.even(3) = ", even.even(3)
print "even.pos_even(2) = ", even.pos_even(2)
print "even.pos_even(3) = ", even.pos_even(3)
print "even.pos_even(-2) = ", even.pos_even(-2)
print "even.pos_even(-3) = ", even.pos_even(-3)
```

## TESTING

We can store the testing statements inside of a module and run them anytime we want to test. But this requires us to manually "check" the correctness of the results.

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

```
$ python test_even.py
even.even(2) = True
even.even(3) = False
even.pos_even(2) = True
even.pos_even(3) = False
even.pos_even(-2) = False
even.pos_even(-3) = False
```

## TESTING

Let's use assert statements to our advantage. Now, when we test, we only need to see if there were any AssertionError exceptions raised. No output means all tests passed!

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

```
import even
assert even.even(2) == True
assert even.even(3) == False
assert even.pos_even(2) == True
assert even.pos_even(3) == False
assert even.pos_even(-2) == False
assert even.pos_even(-3) == False
```

## TESTING

Let's use assert statements to our advantage. Now, when we test, we only need to see if there were any AssertionError exceptions raised. No output means all tests passed!

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

```
$ python test_even.py
$
```

However, one error will halt our testing program entirely so we can only pick up one error at a time. We could wrap assertions into try/except statements but now we're starting to do a lot of work for testing.

There must be a better way!

## UNITTEST

The unittest module in the Standard Library is a framework for writing unit tests, which specifically test a *small* piece of code in isolation from the rest of the codebase.

Test-driven development is advantageous for the following reasons:

- Encourages modular design.
- Easier to cover every code path.
- The actual process of testing is less time-consuming.

## UNITTEST

Here's an example of the simplest usage of unittest.

test\_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

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

even.py

```
def even(num):
    if abs(num) % 2 == 0:
        return True
    return False
```

\$ python test\_even.py

```
.
-----
Ran 1 test in 0.000s

OK
```

## UNITTEST

Here's an example of the simplest usage of unittest.

test\_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

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

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False
```

\$ python test\_even.py

```
.....
 Ran 1 test in 0.000s
OK
```

All tests are defined in methods (which must start with "test\_") of some custom class that derives from unittest.TestCase.

## UNITTEST BEHIND THE SCENES

By calling unittest.main() when we run the module, we are giving control to the unittest module. It will create a new instance of EvenTest for every test method we have so that they can be performed in isolation.

test\_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

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

What unittest does:

```
testcase = EvenTest()

try:
    testcase.test_is_two_even()
except AssertionError:
    [record failure]
else:
    [record success]
```

## UNITTEST

test\_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        self.assertTrue(even.even(2))
    def test_two_positive(self):
        self.assertTrue(even.pos_even(2))

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

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num > 0:
            return False
        return True
    return False
```

We've added some new tests along with some new source code. A couple things to notice: our source code has a logical error in it and we're no longer manually asserting. We're using unittest's nice assert methods.

## UNITTEST

```
$ python test_even.py
.F
```

```
=====
FAIL: test_two_positive (_main_.EvenTest)
-----
Traceback (most recent call last):
  File "test_even.py", line 8, in test_two_positive
    self.assertTrue(even.pos_even(2))
AssertionError: False is not true
=====
Ran 2 tests in 0.000s
```

```
FAILED (failures=1)
```

Extra information given to us by unittest's special assertion method.



## UNITTEST

The unittest module defines a ton of assertion methods:

- `assertEqual(f, s)`
- `assertNotEqual(f, s)`
- `assertIn(f, s)`
- `assertIs(f, s)`
- `assertGreater(f, s)`
- `assertRaises(exc, f, ...)`
- etc.

## UNITTEST

```
test_even.py
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        self.assertTrue(even.even(2))
    def test_two_positive(self):
        self.assertTrue(even.pos_even(2))

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

```
even.py
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

## UNITTEST

```
$ python test_even.py
```

```
..
```

```
-----  
Ran 2 tests in 0.000s
```

```
OK
```

```
$
```

## UNITTEST

We incrementally add unit test functions and run them – when they pass, we add more code and develop the unit tests to assert correctness. Do not remove unit tests as you pass them.

Also, practice unit testing as much as you can. Do not wait until it is absolutely necessary.

As with logging, there is a lot more to unit testing that we're not covering here so definitely look up the docs and read articles about unit testing in Python to learn more about the advanced features.

Ned Batchelder also has a relevant unit testing talk from PyCon '14. Check it out [here](#).

## DOCUMENTATION

Being able to properly document code, especially large projects with multiple contributors, is incredibly important.

Code that is poorly-documented is sooner thrown-out than agonized over. So make sure your time is well-spent and document your code for whoever may need to see it in the future!

Python, as to be expected, has a selection of unique Python-based documenting tools and as a Python developer, it is important that you be familiar with at least one of them.