```
import sys
```

```
>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=6, micro=0, releaselevel='final', serial=0)
>>> sys.version_info.major == 3
True
```

Standard Library

```
import logging
```

stdlib_logging.py

```
import os
import platform
import logging

if platform.platform().startswith('Windows'):
    logging_file = os.path.join(os.getenv('HOMEDRIVE'),
                                os.getenv('HOMEPATH'),
                                'test.log')
else:
    logging_file = os.path.join(os.getenv('HOME'),
                                'test.log')

print("Logging to", logging_file)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s : %(levelname)s : %(message)s',
    filename=logging_file,
    filemode='w',
)

logging.debug("Start of the program")
logging.info("Doing something")
logging.warning("Dying now")
```

```
$ python stdlib_logging.py
Logging to /Users/swa/test.log

$ cat /Users/swa/test.log
2014-03-29 09:27:36,660 : DEBUG : Start of the program
2014-03-29 09:27:36,660 : INFO : Doing something
2014-03-29 09:27:36,660 : WARNING : Dying now
```

Passing tuples around

return two different values from a function

```
>>> def get_error_details():
...     return (2, 'details')
...
>>> errnum, errstr = get_error_details()
>>> errnum
2
>>> errstr
'details'
```

the fastest way to swap two variables in Python

```
>>> a = 5; b = 8
>>> a, b
(5, 8)
>>> a, b = b, a
>>> a, b
(8, 5)
```

Special Methods

Some useful special methods are listed in the following table. If you want to know about all the special methods, see the manual.

docs.python.org/3/reference/datamodel.html#special-method-names

- `__init__(self, ...)`
  - This method is called just before the newly created object is returned for usage.
- `__del__(self)`
  - Called just before the object is destroyed (which has unpredictable timing, so avoid using this)
- `__str__(self)`
  - Called when we use the `print` function or when `str()` is used.
- `__lt__(self, other)`
  - Called when the *less than* operator (<) is used. Similarly, there are special methods for all the operators (+, >, etc.)
- `__getitem__(self, key)`
  - Called when `x[key]` indexing operation is used.
- `__len__(self)`
  - Called when the built-in `len()` function is used for the sequence object.

lambda

more_lambda.py

```
points = [{'x': 2, 'y': 3},
          {'x': 4, 'y': 1}]
points.sort(key=lambda i: i['y'])
print(points)
```

Output:

```
$ python more_lambda.py
[{'y': 1, 'x': 4}, {'y': 3, 'x': 2}]
```

more_list_comprehension.py

```
listone = [2, 3, 4]
listtwo = [2*i for i in listone if i > 2]
print(listtwo)
```

```
$ python more_list_comprehension.py
[6, 8]
```

a special way of receiving parameters to a function as a tuple or a dictionary

```
Def myfunction(*xtuple, **xdictionary)
```

```
>>> def powersum(power, *args):
...     '''Return the sum of each argument raised to the specified power.'''
...     total = 0
...     for i in args:
...         total += pow(i, power)
...     return total
...
>>> powersum(2, 3, 4)
25
>>> powersum(2, 10)
100
```

assert

```
>>> mylist = ['item']
>>> assert len(mylist) >= 1
>>> mylist.pop()
'item'
>>> assert len(mylist) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

The `assert` statement is used to assert that something is true. For example, if you are very sure that you will have at least one element in a list you are using and want to check this, and raise an error if it is not true, then `assert` statement is ideal in this situation. When the assert statement fails, an `AssertionError` is raised. The `pop()` method removes and returns the last item from the list.

The `assert` statement should be used judiciously. Most of the time, it is better to catch exceptions, either handle the problem or display an error message to the user and then quit.

Decorators are a shortcut to applying wrapper functions.

Decorators are a shortcut to applying wrapper functions. This is helpful to "wrap" functionality with the same code over and over again. For example, I created a `retry` decorator for myself that I can just apply to any function and if any exception is thrown during a run, it is retried again, till a maximum of 5 times and with a delay between each retry. This is especially useful for situations where you are trying to make a network call to a remote computer:

How It Works

See:

- http://www.ibm.com/developerworks/linux/library/l-cpdecor.html
- http://toumorokoshi.github.io/dry-principles-through-python-decorators.html

See:
http://www.ibm.com/developerworks/linux/library/l-cpdecor.html

http://toumorokoshi.github.io/dry-principles-through-python-decorators.html

more_decorator.py

```
from time import sleep
from functools import wraps
import logging
logging.basicConfig()
log = logging.getLogger("retry")
```

```
def retry(f):
    @wraps(f)
    def wrapped_f(*args, **kwargs):
        MAX_ATTEMPTS = 5
        for attempt in range(1, MAX_ATTEMPTS + 1):
            try:
                return f(*args, **kwargs)
            except:
                log.exception("Attempt %s/%s failed : %s",
                              attempt,
                              MAX_ATTEMPTS,
                              (args, kwargs))
                sleep(10 * attempt)
        log.critical("All %s attempts failed : %s",
                     MAX_ATTEMPTS,
                     (args, kwargs))
    return wrapped_f


counter = 0
```

```
@retry
def save_to_database(arg):
    print("Write to a database or make a network call or etc.")
    print("This will be automatically retried if exception is thrown.")
    global counter
    counter += 1
    # This will throw an exception in the first call
    # And will work fine in the second call (i.e. a retry)
    if counter < 2:
        raise ValueError(arg)
```

```
if __name__ == '__main__':
    save_to_database("Some bad value")
```

```
$ python more_decorator.py
Write to a database or make a network call or etc.
This will be automatically retried if exception is thrown.
ERROR:retry:Attempt 1/5 failed : (('Some bad value',), {})
Traceback (most recent call last):
  File "more_decorator.py", line 14, in wrapped_f
    return f(*args, **kwargs)
  File "more_decorator.py", line 39, in save_to_database
    raise ValueError(arg)
ValueError: Some bad value
Write to a database or make a network call or etc.
This will be automatically retried if exception is thrown.
```