Brief Tour of the Standard Library

Operating System Interface

The os module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs')   # Change current working directory
>>> os.system('mkdir today')   # Run the command mkdir in the system shell
0
```

Be sure to use the import os style instead of from os import *. This will keep os.open() from shadowing the built-in open() function which operates much differently.

The built-in dir() and help() functions are useful as interactive aids for working with large modules like os:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the shutil module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

String Pattern Matching

The re module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

Mathematics

The math module gives access to the underlying C library functions for floating point math:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The random module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are urllib2 for retrieving data from URLs and smtplib for sending mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

Dates and Times

The datetime module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

Data Compression

Common data archiving and compression formats are directly supported by modules including: zlib, gzip, bz2, zipfile and tarfile.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

Python GUI Programming (Tkinter)

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.

- Tkinter: Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this chapter.
- wxPython: This is an open-source Python interface for wxWindows http://wxpython.org.
- JPython: JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine http://www.jython.org.

There are many other interfaces available, which you can find them on the net.

Tkinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

- Import the *Tkinter* module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

Example

```
#!/usr/bin/python

import Tkinter
top = Tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

This would create a following window −

Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

There are currently 15 types of widgets in Tkinter. We present these widgets as well as a brief description in the following table −

Operator	Description
Button	The Button widget is used to display buttons in your application.
Canvas	The Canvas widget is used to draw shapes, such as lines, ovals, polygons and rectangles, in your application.
Checkbutton	The Checkbutton widget is used to display a number of options as checkboxes. The user can select multiple options at a time.
Entry	The Entry widget is used to display a single-line text field for accepting values from a user.
Frame	The Frame widget is used as a container widget to organize other widgets.

SACET

Label	The Label widget is used to provide a single-line caption for other widgets. It can also contain images.
Listbox	The Listbox widget is used to provide a list of options to a user.
Menubutton	The Menubutton widget is used to display menus in your application.
Menu	The Menu widget is used to provide various commands to a user. These commands are contained inside Menubutton.
Message	The Message widget is used to display multiline text fields for accepting values from a user.
Radiobutton	The Radiobutton widget is used to display a number of options as radio buttons. The user can select only one option at a time.
Scale	The Scale widget is used to provide a slider widget.
Scrollbar	The Scrollbar widget is used to add scrolling capability to various widgets, such as list boxes.
Text	The Text widget is used to display text in multiple lines.
Toplevel	The Toplevel widget is used to provide a separate window container.
Spinbox	The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.
PanedWindow	A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
LabelFrame	A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
tkMessageBox	This module is used to display message boxes in your applications.

Let us study these widgets in detail −

Standard attributes

Let us take a look at how some of their common attributes.such as sizes, colors and fonts are specified.

- Dimensions
- Colors
- Fonts
- Anchors

SACET

- Relief styles
- Bitmaps
- Cursors

Let us study them briefly –

Geometry Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organizing widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid, and place.

- The *pack()* Method - This geometry manager organizes widgets in blocks before placing them in the parent widget.
- The *grid()* Method - This geometry manager organizes widgets in a table-like structure in the parent widget.
- The *place()* Method -This geometry manager organizes widgets by placing them in a specific position in the parent widget.

turtle — Turtle graphics

Introduction

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

Imagine a robotic turtle starting at (0, 0) in the x-y plane. Give it the command `turtle.forward(15)`, and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command `turtle.left(25)`, and it rotates in-place 25 degrees clockwise.

By combining together these and similar commands, intricate shapes and pictures can easily be drawn.

The turtle module is an extended reimplementation of the same-named module from the Python standard distribution up to version Python 2.5.

It tries to keep the merits of the old turtle module and to be (nearly) 100% compatible with it. This means in the first place to enable the learning programmer to use all the commands, classes and methods interactively when using the module from within IDLE run with the `-n` switch.

The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways. Because it uses tkinter for the underlying graphics, it needs a version of Python installed with Tk support.

The object-oriented interface uses essentially two+two classes:

1. The TurtleScreen class defines graphics windows as a playground for the drawing turtles. Its constructor needs a tkinter.Canvas or a ScrolledCanvasas argument. It should be used when turtle is used as part of some application.

   The function Screen() returns a singleton object of a TurtleScreen subclass. This function should be used when turtle is used as a standalone tool for doing graphics. As a singleton object, inheriting from its class is not possible.

All methods of TurtleScreen/Screen also exist as functions, i.e. as part of the procedure-oriented interface.

2. RawTurtle (alias: RawPen) defines Turtle objects which draw on a TurtleScreen. Its constructor needs a Canvas, ScrolledCanvas or TurtleScreen as argument, so the RawTurtle objects know where to draw.

   Derived from RawTurtle is the subclass Turtle (alias: Pen), which draws on "the" Screen instance which is automatically created, if not already present.

   All methods of RawTurtle/Turtle also exist as functions, i.e. part of the procedure-oriented interface.

The procedural interface provides functions which are derived from the methods of the classes Screen and Turtle. They have the same names as the corresponding methods. A screen object is automatically created whenever a function derived from a Screen method is called. An (unnamed) turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen one has to use the object-oriented interface.

Note

In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument *self* which is omitted here.

Overview of available Turtle and Screen methods

Turtle methods

Turtle motion

Move and draw

- forward() | fd()
- backward() | bk() | back()
- right() | rt()
- left() | lt()
- goto() | setpos() | setposition()
- setx()
- sety()
- setheading() | seth()
- home()
- circle()
- dot()
- stamp()
- clearstamp()
- clearstamps()
- undo()
- speed()

Tell Turtle's state

- position() | pos()

- towards()
- xcor()
- ycor()
- heading()
- distance()

Setting and measurement

- degrees()
- radians()

Pen control

Drawing state

- pendown() | pd() | down()
- penup() | pu() | up()
- pensize() | width()
- pen()
- isdown()

Color control

- color()
- pencolor()
- fillcolor()

Filling

- filling()
- begin_fill()
- end_fill()

More drawing control

- reset()
- clear()
- write()

Turtle state

Visibility

- showturtle() | st()
- hideturtle() | ht()
- isvisible()

Appearance

- shape()
- resizemode()
- shapesize() | turtlesize()
- shearfactor()
- settiltangle()
- tiltangle()
- tilt()
- shapetransform()
- get_shapepoly()

Using events

- onclick()
- onrelease()
- ondrag()

Special Turtle methods

- begin_poly()
- end_poly()
- get_poly()
- clone()
- getturtle() | getpen()
- getscreen()
- setundobuffer()
- undobufferentries()

Methods of TurtleScreen/Screen

Window control

- bgcolor()
- bgpic()
- clear() | clearscreen()
- reset() | resetscreen()
- screensize()
- setworldcoordinates()

Animation control

- delay()
- tracer()
- update()

Using screen events

- listen()
- onkey() | onkeyrelease()
- onkeypress()
- onclick() | onscreenclick()
- ontimer()
- mainloop()

Settings and special methods

- mode()
- colormode()
- getcanvas()
- getshapes()
- register_shape() | addshape()
- turtles()
- window_height()
- window_width()

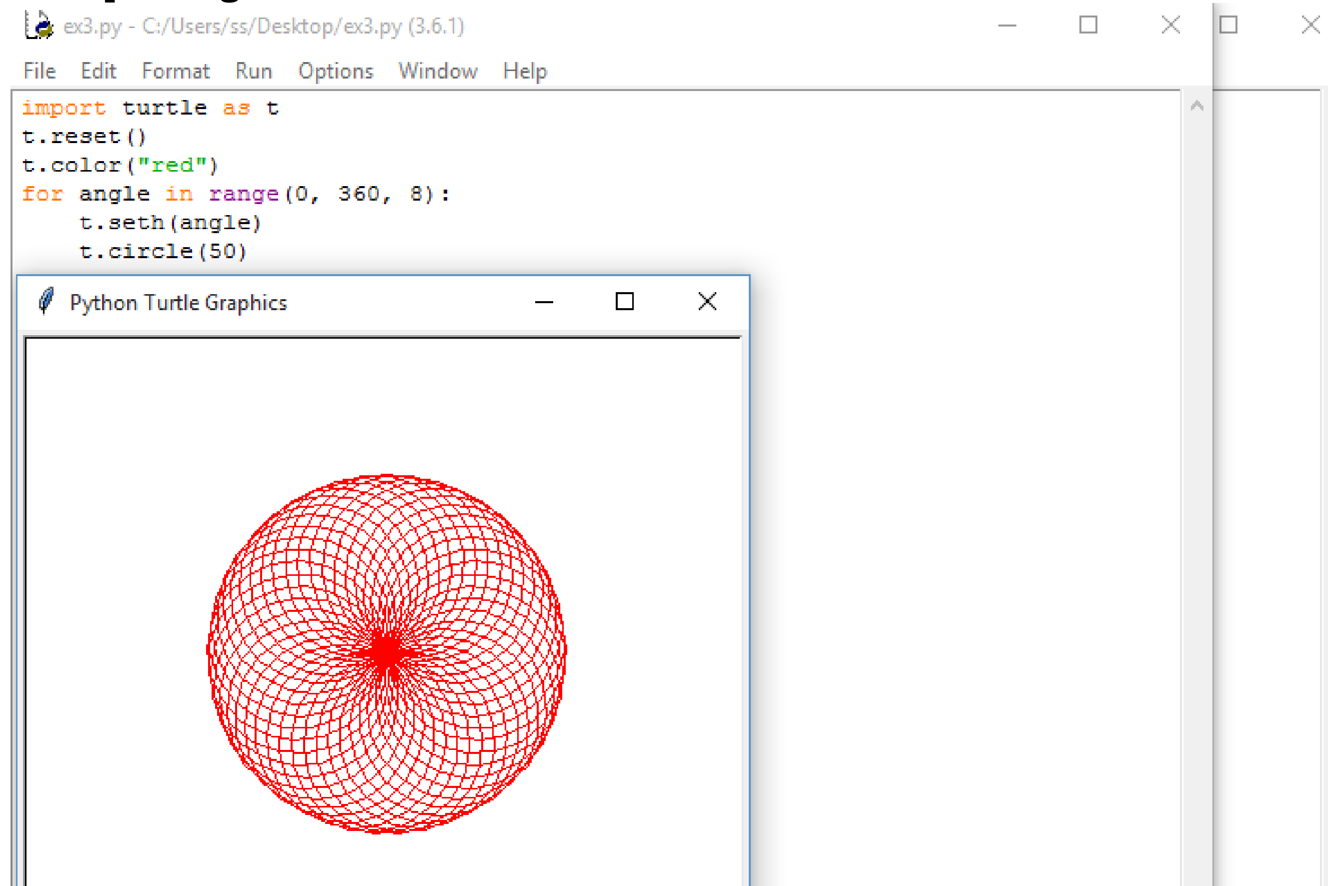
Input methods

- textinput()
- numinput()

Methods specific to Screen

- bye()
- exitonclick()
- setup()
- title()

Example Program:



Why testing is required ?,

Software testing is very important because of the following reasons:

1. Software testing is really required to point out the **defects** and errors that were made during the **development phases**.
2. It's essential since it makes sure of the Customer's reliability and their satisfaction in the application.
3. It is very important to ensure the Quality of the product. Quality product delivered to the customers helps in gaining their confidence. (Know more about **Software Quality**)
4. Testing is necessary in order to provide the facilities to the customers like the delivery of high quality product or software application which requires lower maintenance cost and hence results into more accurate, consistent and reliable results.
5. Testing is required for an effective performance of software application or product.
6. It's important to ensure that the application should not result into any **failures** because it can be very expensive in the future or in the later stages of the development.
7. It's required to stay in the business.

Basic Concepts of Testing

What is Testing?

The process of testing the application to make sure that the application is working according to the requirements.

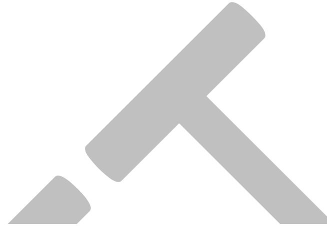
What is Software Testing?

Software testing is the process of executing a program / application under positive and negative conditions by manual or automated means. It checks for the :-

- Specification
- Functionality
- Performance

Who should test?

- Developer
  - Understands the system
  - But,will test gently
- Independent Tester
  - Must learn system
  - But,will attempt to break it



What is an “Error”,”Bug”,”Fault” and “Failure”?

A person makes an Error

That creates a fault in software

That can cause a failure in operation

Error : An error is a human action that produces the incorrect result that results in a fault.

Bug : The presence of error at the time of execution of the software.

Fault : State of software caused by an error.

Failure : Deviation of the software from its expected result. It is an event.

Who is a Software Tester?

Software Tester is the one who performs testing and find bugs, if they exist in the tested application.

The Testing Team

Program Manager

- The planning and execution of the project to ensure the success of a project minimizing risk throughout the lifetime of the project.
- Responsible for writing the product specification, managing the schedule and making the critical decisions and trade-offs.

QA Lead

- Coach and mentor other team members to help improve QA effectiveness
- Work with other department representatives to collaborate on joint projects and initiatives
- Implement industry best practices related to testing automation and to streamline the QA Department.

Test Analyst / Lead

- Responsible for planning, developing and executing automated test systems, manual test plans and regressions test plans.
- Identifying the Target Test Items to be evaluated by the test effort.
- Defining the appropriate tests required and any associated Test Data.
- Gathering and managing the Test Data.
- Evaluating the outcome of each test cycle.

Test Engineer

- Writing and executing test cases and Reporting defects.
- Test engineers are also responsible for determining the best way a test can be performed in order to achieve 100% test coverage of all components.

Unit test — Unit testing framework

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, unittest supports some important concepts in an object-oriented way:

test fixture

A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case

A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class,TestCase, which may be used to create new test cases.

test suite

A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner

A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

Basic example

The unittest module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest

class TestStringMethods(unittest.TestCase):

  def test_upper(self):
      self.assertEqual('foo'.upper(), 'FOO')

  def test_isupper(self):
      self.assertTrue('FOO'.isupper())
      self.assertFalse('Foo'.isupper())
```

```
    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

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

A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letterstest. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to assertEqual() to check for an expected result; assertTrue() or assertFalse() to verify a condition; or assertRaises() to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.

The setUp() and tearDown() methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Passing the -v option to your test script will instruct unittest.main() to enable a higher level of verbosity, and produce the following output:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

The above examples show the most commonly used unittest features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.
Writing test cases

Writing tests for Python is much like writing tests for your own code. Tests need to be thorough, fast, isolated, consistently repeatable, and as simple as possible. We

try to have tests both for normal behaviour and for error conditions. Tests live in the Lib/test directory, where every file that includes tests has a test_ prefix.

One difference with ordinary testing is that you are encouraged to rely on the test.support module. It contains various helpers that are tailored to Python's test suite and help smooth out common problems such as platform differences, resource consumption and cleanup, or warnings management. That module is not suitable for use outside of the standard library.

When you are adding tests to an existing test file, it is also recommended that you study the other tests in that file; it will teach you which precautions you have to take to make your tests robust and portable.

Running test cases

The shortest, simplest way of running the test suite is the following command from the root directory of your checkout (after you have built Python):

```
./python -m test
```

You may need to change this command as follows throughout this section. On most Mac OS X systems, replace ./python with ./python.exe. On Windows, use python.bat. If using Python 2.7, replace test with test.regrtest.

If you don't have easy access to a command line, you can run the test suite from a Python or IDLE shell:

```
>>> from test import autotest
```

This will run the majority of tests, but exclude a small portion of them; these excluded tests use special kinds of resources: for example, accessing the Internet, or trying to play a sound or to display a graphical interface on your desktop. They are disabled by default so that running the test suite is not too intrusive. To enable some of these additional tests (and for other flags which can help debug various issues such as reference leaks), read the help text:

```
./python -m test -h
```

If you want to run a single test file, simply specify the test file name (without the extension) as an argument. You also probably want to enable verbose mode (using -v), so that individual failures are detailed:

```
./python -m test -v test_abc
```

To run a single test case, use the unittest module, providing the import path to the test case:

```
./python -m unittest -v test.test_abc.TestABC
```

If you have a multi-core or multi-CPU machine, you can enable parallel testing using several Python processes so as to speed up things:

```
./python -m test -j0
```

If you are running a version of Python prior to 3.3 you must specify the number of processes to run simultaneously (e.g. -j2).

Finally, if you want to run tests under a more strenuous set of settings, you can run test as:

```
./python -bb -E -Wd -m test -r -w -uall
```

The various extra flags passed to Python cause it to be much stricter about various things (the -Wd flag should be -W error at some point, but the test suite has not reached a point where all warnings have been dealt with and so we cannot guarantee that a bug-free Python will properly complete a test run with -W error). The -r flag to the test runner causes it to run tests in a more random order which helps to check that the various tests do not interfere with each other. The -w flag causes failing tests to be run again to see if the failures are transient or consistent. The -uall flag allows the use of all available resources so as to not skip tests requiring, e.g., Internet access.

To check for reference leaks (only needed if you modified C code), use the -R flag. For example, -R 3:2 will first run the test 3 times to settle down the reference count, and then run it 2 more times to verify if there are any leaks.

You can also execute the Tools/scripts/run_tests.py script as found in a CPython checkout. The script tries to balance speed with thoroughness. But if you want the most thorough tests you should use the strenuous approach shown above.

(for example, the default security settings on some platforms will disallow some tests)

SAC