Importing External Modules

Basic Import Example

```
## 'sys' gives us access to various system details
import sys

## access variables, functions classes defined in module:
print(sys.argv)
print(sys.exc_info())

## multiple layers of namespaces are possible
import os.path
print(os.path.sep)
```

- Access variables, functions, and classes defined in other files
- “system” modules or self-defined

Import Mechanics

When an `import` statement is executed:

1. Code for the module is located
2. List `sys.modules` is checked to see if it is already imported
3. If not already imported, the module code is executed
4. The module object is bound to local name

import … as

```
math = “math is good”

import math as m
print(m.pi)

import numpy as np
import matplotlib.pyplot as plt
```

- Allows the module to be referred to by a different name
- Saves typing (especially when importing multiple layers of namespace)
- Avoids name clashes with existing objects

Import Specific Attributes

```
from math import sin, cos

a = sin(3.14159)

from math import pi as PI
b = cos(PI)
```

- Selectively import variables, functions, and classes
- Convenient and improves readability
- Can also rename with the …`as`… syntax

from … import *

```
## import everything into the current namespace
from math import *

## now (almost) everything is available to us
a = sin(pi)
b = tan(tau)
c = sqrt(e)
```

- Bulk import of the attributes of a module into the current namespace
- Careful! Might overwrite existing names
- Does not import names that begin with _
  - These are considered “private.”

CSV Module Overview

CSV Files

- Comma separated values
- Lowest common denominator data exchange files
- Think: spreadsheet stored as plaintext
- Easy to deal with, can be read/written with many different tools (e.g., Excel, databases, etc.)

Basic CSV Rules

Summary of IETF RFC 4180 guidelines

- Defines ‘text/csv’ MIME type
- Fields/columns separated by comma
- Lines/rows separated by newline
- Fields **may** be quoted with double quotes
- Fields with embedded spaces, delimiters, and newlines **must** be quoted
- Embedded double-quote characters **must** be represented by a pair of them

Python `csv` Module

- Functions for reading and writing csv files
- Support for different “dialects”
  - The standard is very loose.
  - This enables the module to be configured as needed.

Example: write_csv.py

```
import csv

## equations of motion
s = 0        ## position
v = 100      ## velocity, initially upward
G = -9.8     ## gravity, downward
d_t = 0.1    ## deltaT - timestep
t = 0        ## total time

with open('Data.csv', 'wb') as o_file:
    writer = csv.writer(o_file, quoting=csv.QUOTE_ALL)
    writer.writerow(["T", "S", "V"])
    while s >= 0:
        t += d_t
        s = s + v*d_t - 0.5*G*(d_t**2)
        v = v + G * d_t
        writer.writerow([t, s, v])
```

Reading CSV Files

```
import csv

with open('Example.csv', 'r') as f:
    reader = csv.reader(f)
    first = False
    for row in reader:
        if not first: ## skip the first line (header)
            first = True
        else:
            print(row)
```

- Open a file and process it using defaults.
- Each row is a list of values (as `str`).
- The header row must be explicitly handled.

csv.DictReader

```
import csv

with open('Data.csv') as f:
    ## first row of data is used as column names
    reader = csv.DictReader(f)
    for row in reader:
        print("t=%s, s=%s" % (row['T'], row['S']))
```

- `DictReader` uses the first row of data as column names (can be overridden).
- It enhances readability over list indexes.
- `DictWriter` also exists.

Importing Your Own Modules

Importing Our Modules

"Modularizing" is a good idea in general.

- Breaks larger codebase into multiple files
  - And folders
- Increases cohesion, decreases coupling
- Allows for testing module by module

Basic Concept

```python
## constants
CONSTANT = 5

## functions
def some_function():
    pass

## classes
class MyOwnClass:
    pass
```

stuff.py

```python
## get our own code
import stuff

## access whatever we want
a = stuff.CONSTANT
b = stuff.some_function()
c = stuff.MyOwnClass()
```

script.py

[both files in the same folder]

Import(ant) Notes

- Any file of Python code can be imported.
  - File name must be a valid Python identifier
- The first time a module is imported, all the code it contains is executed.
  - No issues for variable, function, and class definitions
  - Other executable code should be wrapped within a function (especially if it produces output)

Use of `__name__`

```
## this is a file made to be either imported, or directly
## executed on its own

def main():
    pass
if __name__ == '__main__':
    main()
```

- `__name__` contains the name of the module when imported.
- If run directly, it contains "`__main__`".
- Above, `main()` must be run explicitly when importing.

Module Hierarchies

Larger software projects are often organized into **module hierarchies.**

- A module named “my_module” can be defined in two ways:
  1. In a script: `my_module.py`
  2. In a folder: `my_module`
     - If this folder contains a script, `__init__.py`, it will be executed the first time the module is imported.
- A folder variant can contain sub-modules.

Submodules

```
> find a
a
a/sub2.py
a/sub1
a/sub1/__init__.py
a/__init__.py
```

- In the situation above, there are three modules: `a`, `a.sub1`, and `a.sub2`.
- The parent, module is initialized before any child modules (only once).

Basics of Software Testing

Fundamental Software Development Question

When is the code “done”?

- When there are no syntax errors?
- When the deadline passes?
- When you’ve read the code and are pretty sure it works?
- When you finish a rigorous mathematical proof that shows it works?

Done When All the Test Cases Pass!

- You could write zero test cases…
  - But might not survive as a software developer
- What is a test case?
  - A set of inputs to some unit of code (think method parameters)
  - An expected output that corresponds to the inputs
- A test case “passes” when the code executes on the inputs and provides the expected answer.

Good Test Cases?

```
## newton's method of finding a square root
import math

EPSILON = 1e-15

def sqrt(c):
    if c < 0:
        return math.nan
    t = c
    while abs(t - c/t) > EPSILON*t:
        t = (c/t + t) / 2.0

    return t
```

- What inputs are a good set of test cases?
- All possible inputs? Good luck with that…

Software Testing Approach

- Software is too complex to prove correct
  - That is, we cannot prove that it does the correct thing for all possible input combinations
- Software testing is aimed at reducing the probability that software is released containing bugs
  - All software contains bugs, so we are trying to eliminate as many as we can with testing
- Engineering trade-off
  - The cost of testing versus the cost of releasing code containing bugs

Cost of Software Bugs

Cost to fix a defect		Time detected				
		Requirements	Architecture	Construction	System test	Post-release
Time introduced	Requirements	1x	3x	5–10x	10x	10–100x
	Architecture	–	1x	10x	15x	25–100x
	Construction	–	–	1x	10x	10–25x

From: http://en.wikipedia.org/wiki/Software_testing

Unit/Integration Testing

- **Unit testing** happens constantly during software construction.
  - Test the “units” in isolation and in small clusters
  - In Python, modules are good “units”
- **Integration testing** happens later.
  - Testing units in combination
- Automation helps.
  - Python’s unittest module

Testing With Python's Unittest Module

`Unittest` Module

Python implementation of Kent Beck's Xunit framework (JUnit, CPPUnit, etc.)

- **Test case:** individual unit of testing, checks response to a specific set of inputs
- **Test fixture:** preparation and cleanup actions needed for one or more test cases
- **Test suite:** collection of test cases or test suites that can be run together
- **Test runner:** component that executes tests and provides output to the user

Basic Usage

```
## from: https://docs.python.org/3.7/library/unittest.html

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())

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

```
> python3 test_string_methods.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s
```

Basic Usage

Some things to note about previous code

- Test cases are organized with a class derived from `unittest.TestCase`.
- Typically, test cases are in a separate script from the code being tested.
- Test cases call the code under test and then check outcomes using a variety of `assert` statements.

Command-line Interface

- `python3 -m unittest test_module1 test_module2`
  - Runs all tests in the specified modules
- `python3 -m unittest test_module.TestClass`
  - Runs all tests in the specified class
- `python3 -m unittest test_module.TestClass.test_method`
  - Runs the specific test method
- `python3 -m unittest tests/test_something.py`
  - Runs the specified file as a module
- `python3 -m unittest -v test_module`
  - Runs the specified module with high “verbosity”
- `python3 -m unittest`
  - Runs “discovery”—finds all test classes and runs them
- `python3 -m unittest -h`
  - Prints command line options to the unittest module

Assertions

Various assertions are available through `TestCase` base class.

- self.assertEqual(A,B)
- self.assertTrue(A)
- self.assertFalse(B)
- …

```
> python3
>>> import unittest
>>> help(unittest)
```

Fixtures

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self): ## executed before each test case
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                'incorrect default size')
    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                'wrong size after resize')

    def tearDown(self): ## executed after each test case
        self.widget.dispose()
```