

Modules and Objects

Importing modules and using objects

D.S. Hwang

Department of Software Science
Dankook University

Outline

Importing Modules

Defining Your Own Modules

Objects and Methods

Pixels and Colors

Testing

Style Notes

Overview

- ▶ Mathematicians build their proofs on the facts their predecessors have already established.
- ▶ Program designer make use of the millions of lines of code that other programmers have written before.
- ▶ A `module` is a collection of functions that are grouped together in a single file.
 - ▶ Functions in a module are usually related to each other in some way.
 - ▶ The `math` module contains mathematical functions such as `cos` and `sqrt`.

Goals of the chapter

- ▶ How to use some of the hundreds of modules that come with Python
- ▶ How to create new modules of your own

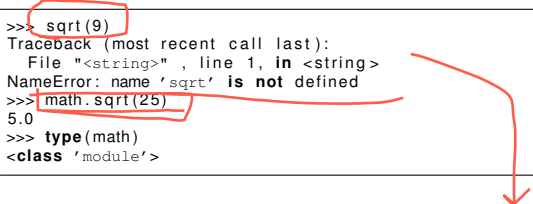
Importing Modules

```
1 >>> import math ✓
2 >>> help(math)
3 Help on built-in module math:
4 NAME
5     math
6 FILE
7     (built-in)
8 DESCRIPTION
9     This module is always available. It provides access to the
10    mathematical functions defined by the C standard.
```

- ▶ Import it when you want to use a function from a module
- ▶ Get the usage of the function by built-in `help` function

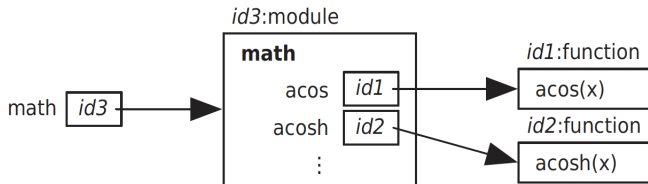
Importing Modules I

```
1 >>> sqrt(9)
2 Traceback (most recent call last):
3   File "<string>", line 1, in <string>
4 NameError: name 'sqrt' is not defined
5 >>> math.sqrt(25)
6 5.0
7 >>> type(math)
8 <class 'module'>
```



- ▶ We get an error telling us that Python is unable to find the function `sqrt`.
- ▶ Python can look for the function in the `math` module by combining the module's name with the function's name using a dot.
- ▶ Importing a module creates a new variable with that name.
- ▶ That variable refers to an object whose type is `module`.

Importing Modules II

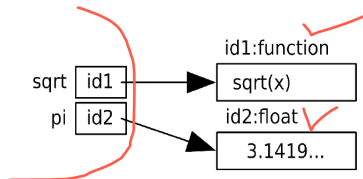


```
1 >>> id (math)
2 4416144072
3 >>> id (math.cos)
4 4416138984
5 >>> id (math.acos)
6 4416136680
7 >>> id (math.acosh)
8 4416136752
9 >>> id (math.sin)
10 4416141136
```

Importing Modules III

```
1 >>> from math import sqrt, pi ✓
2 >>> sqrt(9)
3 3.0
4 >>> radius = 5
5 >>> print('circumference is', 2 * pi * radius)
6 circumference is 31.41592653589793
7 >>> math.sqrt(9)
8 Traceback (most recent call last):
9   File "<pyshell#12>", line 1, in <module>
10    math.sqrt(9)
11 NameError: name 'math' is not defined
12 >>> sqrt(9)
13 3.0
```

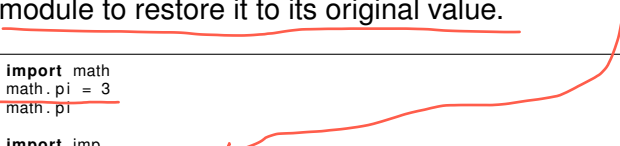
- ▶ Doesn't introduce a variable called `math`.
- ▶ It creates function `sqrt` and variable `pi` in the *current namespace*



Restoring Modules

- ▶ We change the value of a variable or function from an imported module.
- ▶ For restoration, we can restart the shell or reimport the module to restore it to its original value.

```
1 >>> import math
2 >>> math.pi = 3
3 >>> math.pi
4 3
5 >>> import imp
6 >>> math = imp.reload(math)
7 >>> math.pi
8 3.141592653589793
```



Importing Modules

Why do we have to join the function's name with the module's name?

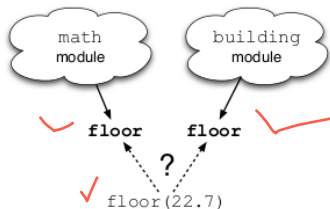


Figure: How import works

- ▶ Several modules might contain functions with the same name.
- ▶ Once a module has been imported, it stays in memory until the program ends.

Importing Modules

Modules contain functions and variables.

```
1 >>> math.pi
2 3.1415926535897931
3 >>> radius = 5
4 >>> print('area is %6f' % (math.pi * radius ** 2))
5 area is 78.539816
```

✓ You can change the variables in the module.

```
1 >>> import math
2 >>> math.pi = 3
3 >>> radius = 5
4 >>> print('circumference: ', 2*math.pi*radius)
5 circumference: 30
```

Importing Modules

Python lets you specify exactly what you want to import from a module

```
1 >>> from math import sqrt, pi
2 >>> sqrt(9)
3 3.0
4 >>> radius = 5
5 >>> print('circumference is %6f' % (2 * pi * radius))
6 circumference is 31.415927
```

Python lets you import all the functions within a module

```
1 >>> from math import *
2 >>> '%6f' % sqrt(8)
3 2.828427
```

Importing Modules

- ▶ The standard Python library contains several hundred modules.
- ▶ The full list is online at <http://docs.python.org/modindex.html>

Defining Your Own Modules

- ▶ Every Python file can be used as a module.
- ▶ A Python module ends with a .py extension.
- ▶ The name of the module is the same as the name of the file, but without the .py extension.

```
1 #!/usr/bin/env python
2 # temperature.py
3 def to_celsius(t):
4     return (t - 32.0) * 5.0 / 9.0
5
6 def above_freezing(t):
7     return t > 0
```

Defining Your Own Modules

Import temperature.py:

```
1 >>> from temperature import *
2 >>> temperature.above_freezing(to_celsius(33.3))
3 True
```

- ▶ Import it when you want to use a function from a module
- ▶ Get the usage of the function by built-in help function

Defining Your Own Modules

`__builtins__` module

- ▶ provides Python's built-in functions
- ▶ `help(__builtins__)` checks what's in the module and how to call them.
- ▶ `dir(__builtins__)` checks a directory.

```
1 >>> help(__builtins__)
2 ...
3 >>> dir(__builtins__)
4 >>> dir(__builtins__)
5 ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'By
```

Defining Your Own Modules

What Happens when importing a module:

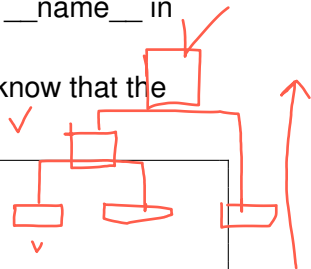
```
1 % cat experiment.py
2 print("The panda's scientific name is ' Ailuropoda melanoleuca'")
3 % python
4 >>> import experiment
5 The panda's scientific name is ' Ailuropoda melanoleuca'
```

- ▶ Python executes modules as it imports them.
- ▶ Python loads modules only the first time they are imported

Defining Your Own Modules

Using __name__

- ▶ Every Python file can be run directly from the command line or IDE or can be imported and used by another program.
- ▶ Python defines a special variable called __name__ in every module.
- ▶ The __name__ variable tells Python to know that the module is the main program.



```
1 % cat echo.py
2 print("echo: __name__ is", __name__)
3 % python echo.py
4 echo: __name__ is __main__
5 % python
6 ...
7 >>> import echo.py # __main__ is not '__main__'
8 echo: __name__ is echo
9 >>> __name__
10 '__main__'
```

Defining Your Own Modules

Using `__name__`


- ▶ When Python imports a module, it sets that module's `__name__` variable to be the name of the module, rather than the special string `"__main__"`.
- ▶ A module can tell whether it is the main program:

```
1 % cat test_main.py
2 if __name__ == "__main__":
3     print("I am the main program")
4 else:
5     print("Someone is importing me")
6 % python test_main.py
7 I am the main program
8 % python
9 >> import test_main.py
10 Someone is importing me
```

Defining Your Own Modules

Glance at a standard Python structure:

```
1  '''
2  This module guesses whether something is a dinosaur
3  or not.
4  '''
5
6  def is_dinosaur(name):
7      '''
8      Return True if the named creature is recognized as
9      a dinosaur, and False otherwise.
10     '''
11     return name in ['Tyrannosaurus', 'Triceratops']
12
13 if __name__ == '__main__':
14     help(__name__)
```



Defining Your Own Modules

Try the mentioned Python code:

```
1 Help on built-in module __main__:
2
3 NAME
4     __main__ - This module guesses whether something is a dinosaur or not.
5
6 FILE
7     /home/dshwang/dsLecture/Markup/python-pgm/main_help.py
8
9 FUNCTIONS
10     is_dinosaur(name)
11         Return True if the named creature is recognized as a dinosaur,
12         and False otherwise.
```

Defining Your Own Modules I

Learn the easy way to comment Python codes:

```
1 % cat temp_round.py
2 def to_celsius(t):
3     return round((t - 32.0) * 5.0 / 9.0)
4 def above_freezing(t):
5     return t > 0
```

Defining Your Own Modules II

Check out the result:

```
1 Help on module temp_round:
2
3 NAME
4     temp_round
5
6 FILE
7     /home/dshwang/dsLecture/Markup/python-pgm/temp_round.py
8
9 FUNCTIONS
10    ✓ above_freezing(t)
11
12    ✓ to_celsius(t)
```

♣ We know the names of the functions and how many parameters they need.

Defining Your Own Modules I

- docstrings makes the module and the function much more useful.

Learn the easiest way to comment Python codes:

```
1 % cat temp_round.py
2 '''Functions for working with temperatures.'''
3
4 def to_celsius(t):
5     '''Convert Fahrenheit to Celsius.'''
6
7     return round((t - 32.0) * 5.0 / 9.0)
8
9 def above_freezing(t):
10     '''True if temperature in Celsius is above freezing ,
11        False otherwise.'''
12
13     return t > 0
```

Defining Your Own Modules II

Check out the result:

```
1 Help on module temp_round:
2
3 NAME
4     temp_round — Functions for working with temperatures.
5
6 FILE
7     /home/dshwang/dsLecture/Markup/python-pgm/temp_round.py
8
9 FUNCTIONS
10    above_freezing(t)
11        True if temperature in Celsius is above freezing ,
12        False otherwise .
13
14    to_celsius(t)
15        Convert Fahrenheit to Celsius.
```


Defining Your Own Modules

- ▶ The term `docstring` is short for “documentation string.”
- ▶ If the first thing in a file or a function is a string that isn't assigned to anything, Python saves it so that help can print it later.
- ▶ The use of `docstring` makes software manageable.

High-level Documentation with Python

Doc string:

- ▶ Python strings that appear at special locations
- ▶ Act as user documentation of modules, classes, and functions.
- ▶ Python programming style guide recommends triple double quoted strings as doc strings.
- ▶ Provide a unified way of explaining the purposes and usage of modules, classes, and functions.

▶ Placement:

Modules Doc strings appear as the first string.

Class Doc strings appear right after the class name.

Function Doc strings appear right after the function heading.

▶ Get the guide:

```
1 module_name.__doc__  
2 class_name.__doc__  
3 function_name.__doc__
```

Automatic Generated Documentation

Tools:


Doxygen <http://www.doxygen.org>

HappyDoc <http://happydoc.sourceforge.net>

Epydoc <http://epydoc.sourceforge.net>

Pydoc produces a HTML documentation(default tool).

```
% pydoc -w circle.py
```



Automatic Generated Documentation

```
1  """ Module: circle.py
2      circle module: contains the Circle class.
3  """
4  class Circle:
5      """Circle class"""
6      r = 0;
7      pi = 3.14159;
8      def __init__(self, r=1):
9          """create a Circle with the given radius"""
10         self.radius = r
11
12     def area(self):
13         """determine the area of the Circle"""
14         return self.__class__.pi * self.radius * self.radius
15
16 if __name__ == "__main__":
17     """ do a unit test """
18     circle = Circle( 5.0 );
19     val = circle.area();
20     print("Area is " + str(val));
```

Automatic Generated Documentation



Figure: Automatic documentation through Pydoc

Objects and Methods

Kinds of data we meet:

- ▶ We are dealing with the popular data of numbers and strings.
- ▶ These days, we expect to work with images, sound, and video as well

A Python provides functions for manipulating and viewing pictures.

- ▶ No standard library
- ▶ PyGrahics, <http://code.google.com/p/pygraphics/>

Objects and Methods

Two fundamental concepts to modern program design:

- ✓ 1. single-character operators
 - ▶ Two same operators that work on strings and numbers: +
 - ▶ Formatter on strings: % ✓
 - ▶ Operators: +, -, ×, ÷ ✓
- 2. Python strings “own” a set of special functions called methods.
 - ▶ All the functions that work on strings in a module and ask users to load that module
 - ▶ Python strings “own” a set of special functions called methods.
 - ▶ Every string we create automatically shares all the methods that belong to the string data type.

functionName

Objects and Methods

```
1 >>> s = 'hogwarts'
2 >>> s.capitalize()
3 'Hogwarts'
4 >>> s = 'hogwarts'
5 >>> s.capitalize()
6 'Hogwarts'
7 >>> help(str)
8 ...
```


Objects and Methods

- ▶ Something that has methods is called an object.
- ▶ **Everything in Python is an object.**

```
1 >>> help(0)
2 Help on int object:
3
4 class int(object)
5 |     int(x[, base]) -> integer
6 |
7 |     Convert a string or number to an integer, if possible.
8 |
9 |     __abs__(...)
10 |    x.__abs__() <==> abs(x) ✓
11 |
12 |     __add__(...)
13 |    x.__add__(y) <==> x+y ✓
14 |
15 |     __and__(...)
16 |    x.__and__(y) <==> x&y ✓
```

Objects and Methods

The basic structure of most modern programming languages:

The “things” in the program are objects, and most of the code in the program consists of methods that use the data stored in those objects.

- ▶ This basis helps us to understand Object-Oriented Programming.

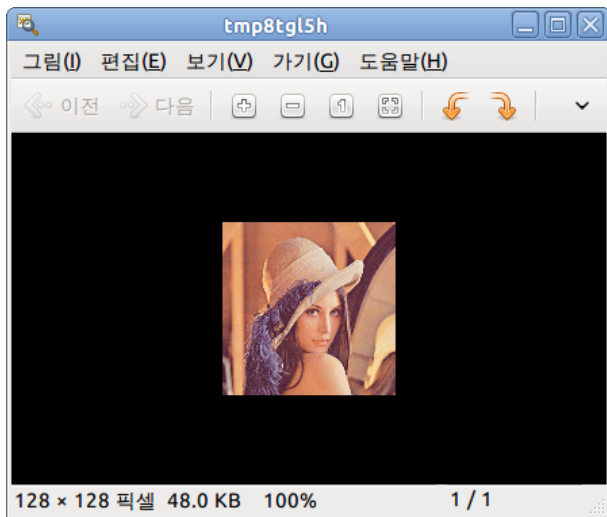
Objects and Methods

Images

- ▶ A program that displays and manipulates pictures and other images.

```
1 >>> from PIL import Image
2 >>> im = Image.open("lena.jpg")
3 >>> im
4 <JpegImagePlugin.JpegImageFile instance at 0xb7338acc>
5 >>> print( im.format, im.size, im.mode)
6 JPEG (128, 128) RGB
7 >>> im.show(title="Lena", command="eog")
```

Objects and Methods



Objects and Methods

Images

- ▶ A program that displays and manipulates pictures and other images.

```
1 >>> from PIL import Image
2 >>> im = Image.open("lena.jpg")
3 >>> im
4 <JpegImagePlugin.JpegImageFile instance at 0xb7338acc>
5 >>> print( im.format, im.size, im.mode)
6 JPEG (128, 128) RGB
7 >>> im.show(title="Cozy House", command="eog")
```

Objects and Methods



Pixels and Colors

- ▶ The `Image` module represents pixels using the RGB color model.
- ▶ The `Image` provides a `Color` type and more than 100 predefined `Color` values (see Figure 4.3).
- ▶ The `Image` module provides functions for getting and changing the colors in pixels (see Figure 4.9).

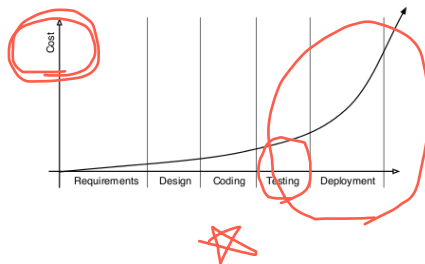
Testing

Quality assurance, QA

- ▶ Modular design is more advantageous in solving a computational problem.
- ▶ Quality assurance checks that software is doing the right thing.
- ▶ Quality must be considered at the design time, and software must be tested and retested to check that it meets standards.
- ▶ Putting effort into QA actually makes you more productive overall(see Boehm's cure).

Testing

Most good programmers today don't just test their software while writing it; they build their tests so that other people can rerun them months later and a dozen time zones away. This takes a little more time up front, but makes programmers more productive overall, since every hour invested in preventing bugs saves two, three, or ten frustrating hours tracking bugs down.



Testing

Nose

- ▶ One popular testing library for Python
- ▶ <http://code.google.com/p/python-nose/>
- ▶ Nose recognizes the file with `test_fn.py`.
 - ▶ Statements to import Nose and the module to be tested
 - ▶ Functions that actually test our module
 - ▶ A function call to trigger execution of those test functions

```
1 $ python test_temperature.py
2 ..
3 -----
4 Ran 2 tests in 0.001s
5
6 OK
```

Testing

```
1 $ cat test_temperature.py
2 import nose ✓
3 import temperature
4
5 def test_to_celsius():
6     '''Test function for to_celsius'''
7     pass # we'll fill this in later
8
9 def test_above_freezing():
10    '''Test function for above_freezing.'''
11    pass # we'll fill this in too
12
13 if __name__ == '__main__':
14    nose.runmodule()
```

Testing

```
1 $ cat assert.py
2 import nose
3 from temp_with_doc import to_celsius
4 def test_freezing():
5     '''Test freezing point.'''
6     assert to_celsius(32) == 0 # 32 C = 0 F
7 def test_boiling():
8     '''Test boiling point.'''
9     assert to_celsius(212) == 100 # 212 C= 100 F
10 def test_roundoff():
11     '''Test that roundoff works.'''
12     assert to_celsius(100) == 38 # NOT 37.777...
13 if __name__ == '__main__':
14     nose.runmodule()
```

Testing

```
1 $ pyhton assert.py
2 ...
3
4 Ran 3 tests in 0.001s
5
6 OK
```

One of three outcomes

- ✓ ➤ Pass. The actual value matches the expected value.
- ✓ ➤ Fail. The actual value is different from the expected value.
- ✓ ➤ Error. Something went wrong inside the test itself; in other words, the test code contains a bug. In this case, the test doesn't tell us anything about the system being tested.

Testing

```
1 $ cat assert2.py
2 import nose
3 from temp_with_doc import to_celsius
4 def test_freezing():
5     '''Test freezing point.'''
6     assert to_celsius(32) == 0
7 def test_boiling():
8     '''Test boiling point.'''
9     assert to_celsius(212) == 100
10 def test_roundoff():
11     '''Test that roundoff works.'''
12     assert to_celsius(100) == 37.8
13 if __name__ == '__main__':
14     nose.runmodule()
```

Testing

```
1 $ python assert2.py
2 ..F
3 =====
4 FAIL: Test that roundoff works.
5 -----
6 Traceback (most recent call last):
7   File "/usr/lib/pymodules/python2.6/nose/case.py", line 183, in runTest
8     self.test(*self.arg)
9   File "assert2.py", line 14, in test_roundoff
10    assert to_celsius(100) == 37.8
11 AssertionError
12 -----
13
14 Ran 3 tests in 0.017s
15
16 FAILED (failures=1)
```

Testing

```
1 $ cat assert2.py
2 import nose
3 from temp_with_doc import to_celsius
4 def test_freezing():
5     '''Test freezing point.'''
6     assert to_celsius(32) == 0
7 def test_boiling():
8     '''Test boiling point.'''
9     assert to_celsius(212) == 100
10 def test_roundoff():
11     '''Test that roundoff works.'''
12     assert to_celsius(100) == 37.8, 'Returning an unrounded result'
13
14 if __name__ == '__main__':
15     nose.runmodule()
```


Testing

```
1 $ python assert2.py
2 ..F
3 =====
4 FAIL: Test that roundoff works.
5
6 Traceback (most recent call last):
7   File "/usr/lib/pymodules/python2.6/nose/case.py", line 183, in runTest
8     self.test(*self.arg)
9   File "assert2.py", line 14, in test_roundoff
10     assert to_celsius(100) == 37.8, 'Returning an unrounded result'
11 AssertionError
12
13 Ran 3 tests in 0.017s
14 FAILED (failures=1)
```

Testing

```
1 $ cat test_freezing.py
2 import nose
3 from temp_with_doc import above_freezing
4
5 def test_above_freezing():
6     '''Test function for above_freezing.'''
7     assert above_freezing(89.4), 'A temperature above freezing.'
8     assert not above_freezing(-42), 'A temperature below freezing.'
9     assert not above_freezing(0), 'A temperature at freezing.'
10 if __name__ == '__main__':
11     nose.runmodule()
```

Testing

```
1 $ python test_freezing.py
2 .
3 -----
4 Ran 1 test in 0.001s
5
6 OK
```

- ▶ Nose believes that only one test was run, even though there are three assert statements in the file.

Style Notes

- ▶ Anything that can go in a Python program can go in a module.
- ▶ If you have functions and variables that logically belong together, you should put them in the same module.
- ▶ You have to decide based on how more experienced programmers have organized modules like the ones in the Python standard library and eventually on your own sense of style.

Summary










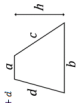

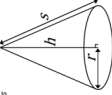


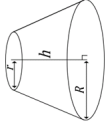
- ▶ A module is a collection of functions and variables grouped together in a file.
- ▶ Put docstrings at the start of modules or functions to describe their contents and use.
- ▶ Every “thing” in a Python program is an object.
- ✓ ▶ Manipulate images using the PIL module
- ▶ Programs have to do more than just run to be useful.
 - ▶ One way to ensure that they do is to test them.
 - ▶ Python uses the Nose module to test your modules.

Problem 1 I

There are various geometries. Design modules to calculate area, perimeter, volume, or surface related to a chosen geometry.

- ▶ Divide the related operations into two groups or more
- ▶ Design and test the designed module
- ▶ Design the related functions by function
- ▶ Write some informative comment to every function
- ▶ Design your drive module for user's input using `if-then` statements

Problem 1 II

GEOMETRY		SHAPES AND SOLIDS	
SQUARE $P = 4s$ $A = s^2$ 	RECTANGLE $P = 2a + 2b$ $A = ab$ 	CIRCLE $P = 2\pi r$ $A = \pi r^2$ 	
TRIANGLE $P = a + b + c$ $A = \frac{1}{2}bh$ 	PARALLELOGRAM $P = 2a + 2b$ $A = bh$ 	CIRCULAR SECTOR $L = \frac{\pi r^2 \theta}{180^\circ}$ $A = \pi r^2 \frac{\theta}{360^\circ}$ 	
PYTHAGOREAN THEOREM $a^2 + b^2 = c^2$ $c = \sqrt{a^2 + b^2}$ 	CIRCULAR RING $A = \pi(R^2 - r^2)$ 	SPHERE $S = 4\pi r^2$ $V = \frac{4\pi r^3}{3}$ 	
TRAPEZOID $P = a + b + c + d$ $A = h \frac{a+b}{2}$ 	RECTANGULAR BOX $A = 2ab + 2ac + 2bc$ $V = abc$ 	RIGHT CIRCULAR CONE $A = \pi r^2 + \pi r s$ $S = \sqrt{r^2 + h^2}$ $V = \frac{1}{3} \pi r^2 h$ 	
CUBE $A = 6l^2$ $V = l^3$ 	CYLINDER $A = 2\pi r(r + h)$ $V = \pi r^2 h$ 	FRUSTUM OF A CONE $V = \frac{1}{3} \pi h (r^2 + rR + R^2)$ 	
<div> <div>EEWeb.com</div> <div>Electrical Engineering Community</div> <ul style="list-style-type: none"> • Latest News • Professional Networking • Engineering Community • Personal Profiles and Resumes • Online Toolbox • Community Blogs and Projects • Technical Discussions • Find Jobs and Events </div>			
EEWeb.com		The Best Source for Electrical Engineering Resources	
EEWeb.com		EEWeb.com	

Problem 2

Given two d -dimensional vectors u and v , the ways to compute their distance are various.

1. Survey more than 5 distance measures between u and v
2. Write a module to compute a distance between two input vector

3. The input vector can be generate by

```
np.random.randint(), np.random.rand()
```

scipy-function

test_main