Computational Physics

User Defined Modules

Jan 31, 2019

http://hadron.physics.fsu.edu/~eugenio/comphy/ eugenio@fsu.edu

pydoc

Documentation generator and online help system

pydoc numpy.random.random

💿 💿 🏠 eugenio — less « python3.7 ~/anaconda3/bin/pydoc numpy.random.random — 8
ogin.rcc.fsu.edu -l eugenionumpy.random.randomes/vpython7 — -bash
Help on built-in function random_sample in numpy.random:
<pre>numpy.random.random = random_sample() method of mtrand.RandomState instance random_sample(size=None)</pre>
Return random floats in the half-open interval [0.0, 1.0).
Results are from the "continuous uniform" distribution over the stated interval. To sample :math:`Unif[a, b), b > a` multiply the output of `random_sample` by `(b-a)` and add `a`::
(b - a) * random_sample() + a
Parameters size : int or tuple of ints, optional Output shape. If the given shape is, e.g., ``(m, n, k)``, then
``m * n * k`` samples are drawn. Default is None, in which case a single value is returned.
Returns
out : float or ndarray of floats Array of random floats of shape `size` (unless ``size=None``, in which case a single float is returned).
Examples
>>> np.random.random_sample() 0.47108547995356098
<pre>>>> type(np.random.random_sample()) <type 'float'=""></type></pre>
>>> np.random.random_sample((5,)) array([0.30220482, 0.86820401, 0.1654503 , 0.11659149, 0.54323428])

Three-by-two array of random numbers from [-5, 0):

Why modules

- Sometimes you want to reuse a function or several functions from an old program in a new program.
 - One could simply copy and paste the old code into the new program.
- The problem with this is that over time you could end up with many copies of the same code
 - if you fix or improve part of the code in one version, you will have to update all copies
 - Or you will end up with multiple versions
 - some useful, some less useful, and possible some which are buggy or faulty

Making Modules

Golden Rule

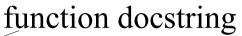
Have one and only one version of a piece of code

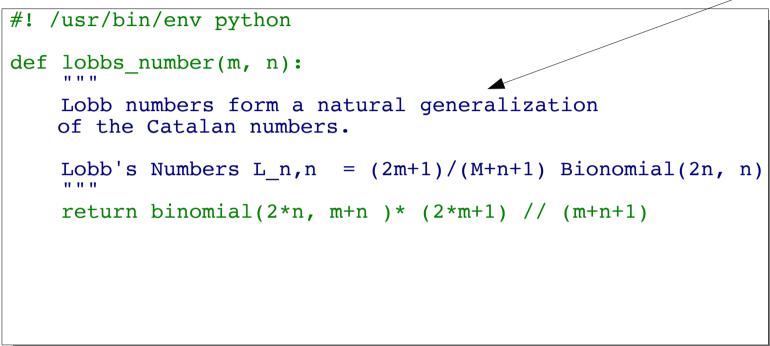
This is easy to implement if we create a module containing the code we want to reuse.

import mystuff

value = mystuff.myfunction(10)

Example: lobbs_number()





We want to make this function available in a module named **mystuff**

import mystuff as my
my.lobbs_number(m, n)

So how do we create the **mystuff** module?

Collecting functions in a module

- Simply create a new source file and copy all of the code into this file.
- Save the file with the module name along with the standard ".py" file extension.

In our case, the filename mystuff.py implies a module with the name mystuff.

Using functions in a module

import mystuff as my lobbs_1_3 = my.lobbs_number(1, 3)

But Python needs to now about the module in order to use it.

 The program which imports you module(s) will work fine if it is located in the same directory as your module

 The program which imports you module(s) will work fine if it is located in the same directory as your module

 However if you move your program to another directory, running the program will give an error.

```
hpc-login 515% lobbs.py
Traceback (most recent call last):
   File "lobbs.py", line 18, in <module>
        import mystuff as my
ImportError: No module named mystuff
```

A better solution is to store your module(s) in your Python search path

A better solution is to store your module(s) in your Python search path

Create a dir/ for storing your python modules

mkdir \$HOME/python/mymodules/

Place your module(s) in this directory

Set the PYTHONPATH environmental variable

cshell command:

setenv PYTHONPATH "\${HOME}/python/mymodules:./mymodules"

bash command:

Add the above path definition to your .cshrc (or .bashrc) file so that \$PYTHONPATH is defined every time you log in.

Set the PYTHONPATH environmental variable

 cshell command: setenv PYTHONPATH "\${HOME}/python/mymodules"

 bash command: export PYTHONPATH=\$HOME/python/mymodules/

Add the above path definition to your **.cshrc** (or .bashrc) file so that **\$PYTHONPATH** is defined every time you log in.

hpc-login-25% emacs ~/.cshrc &

do it!

Doc strings in modules

Always include a *useful* doc string at the beginning of the module.

11 11 11

Module MyStuff is a collection of useful functions which are user defined, stored locally at \$HOME/python/mymodules/mystuff.py where the mymodules directory has been added to the \$PYTHONPATH environment.

Symbols:

'n' is positive integer index

Paul Eugenio Florida State University Department of Physics Jan 2019

11 11 11

header docstring + function docstring

Documentation from Doc Strings



hpc-login 515% python
...
>>> help("mystuff")

You can also run **pydoc** on the module to see the documentation of the new module

hpc-login 515% pydoc mystuff

- • •
- • •
- • •

pups — ssh -Y hpc-login.rcc.fsu.edu -l eugenio — 82×38

....ogin.rcc.fsu.edu -l eugenio

ginn constructur - reugenio

~ — -bash

...es/vpython7 — -bash

Help on module mystuff:

NAME

mystuff

FILE

/gpfs/home/eugenio/python/mymodules/mystuff.py

DESCRIPTION

Module MyStuff is a collection of useful functions which are user defined, stored locally at \$HOME/python/mymodules/mystuff.py where the mymodules directory has been added to the \$PYTHONPATH environment.

Symbols:

'm' is positive integer 'n' is a positive integet such that n >= m >= 0

Paul Eugenio Florida State University Department of Physics Jan 2019

FUNCTIONS

binomial(n, k)
Binomial coefficient (n k) = n!/(k!(n-k)!

factorial(...)

factorial(x) -> Integral

Find x!. Raise a ValueError if x is negative or non-integral.

lobb_number(m, n)

Lobb numbers form a natural generalization of the Catalan numbers,

:

Test block

During import, the module file is fully executed

- The module should have function definitions and should not have any open statements
- It is desirable to have some test or verification code in the module

Test block

During import, the module file is fully executed

- The module should have function definitions and should not have any open statements
- It is desirable to have some test or verification code in the module
- Python allows the file to act both as a module and as a main program
 - To seamlessly have both functionality the main program statements should be in a <u>test block</u>

<block of statements>

Test block

```
# test functions
def test functions():
   Routines to test module functions. To execute test of function run
   module as python program along with command line argument "test"
        example: "mystuff test"
    ......
   # test lobb number function
    if (lobb number(1, 3) == 9):
        print("Module is Good")
    else:
        print("WARNING!!\n lobb number() function failed test\n DO NOT USE!!"
# TEST BLOCK
# The test block only executes if the module is run as a main program
# and if the word "test" is given on the command line.
                                              import sys
if
    name == ' main ':
    if len(sys.argv) == 2 and sys.argv[1] == 'test':
        test functions()
```

hpc-login 515% mystuff.py test Module is Good

Example: User Defined Module

See examples: mystuff.py & lobbs.py

```
#! /usr/bin/env python
# Generate Lobb's Triangle
 this program uses a user defined module mystuff
#
#
# Paul Eugenio
# PHZ4151C
# Jan 31, 2019
from future import division, print function
import mystuff as my
import sys
#set triangle size
if len(sys.argv) == 2:
    size = int(sys.argv[1])
else:
    size = 5
# print out a triangle of Lobb's Numbers
for n in range(size):
   for m in range(n+1):
      print(my.lobb number(m, n), end="\t")
  print()
```

We will soon be covering numerical integration

```
#! /usr/bin/env python
```

```
def trapezoidal(fun, a, b, N):
    ...
def simpson(fun, a, b, N):
    ...
def adapatrap(fun, a, b, N, accuracy):
    ...
def adapasimp(fun, a, b, N, accuracy):
    ...
def mcintegrate(func, dim, limit, N=100):
    ...
```

You will be required to make your own functions available in a module

```
import myintegrate as myint
myint.trapezoidal(f, 0, 1, 100)
```

Doc strings from modules

hpc-login 515% python
...
>>> help("myintegration")

You can also run **pydoc** on the module to see the documentation of the new module

hpc-login 515% pydoc myintegration

Help on module integrate:

NAME

integrate - Module for calculating integrals numerically.

FILE

/panfs/storage.local/physics/home/eugenio/python/exercises/ex5/integrate.py

DESCRIPTION

Symbols:

Series integration (one dimensional)
'fun' is a user defined 1D function
'a' and 'b' are lower and upper integration limits
'N' is the number of steps used in the series integration
'accuracy' is the desired accurcy for adaptive integration.

Monte Carlo mean value integration (any dimensional)
'func' is a user define function of any dimension
'dim' is the dimension of the integrand
'limit' is a list of [a,b] integration limit values
'N' is the number of sampling points (default value = 100 points)

FUNCTIONS

adapasimp(fun, a, b, accuracy)
Adaptive intgration use Simpson's rule. This
method doubles the number of steps but
calculates next integral using the minimum number
terms.

adapatrap(fun, a, b, accuracy)
Adaptive intgration use trapazodial rule. This
method doubles the number of steps but
calculates next integral using the minimum number
terms.

mcintegrate(func, dim, limit, N=100)
Monte carlo mean-value integration of any dimension.
Func is the user defined integrand function
which has a list argument x containing dim values.

for example: sin(x*y) is f(x)= sin(x[0]*x[1])

Dim is the number of dimensions, limit is a list of [a,b] values containg the intgration limits for all dimensions, and N is the number of Monte carlo sampling points.

function returns [result, error]

simpson(fun, a, b, N)
Integration by Simpson's rule using N steps