Functions and Classes

Computational Methods, Oct. 2017, Kenji Doya

Let us learn how to define your own functions, and further organize them into a *class* for neatness and extensibility.

References:

- Python Tutorial section 4.6-4.8: Functions
- Python Tutorial chapter 6: Modules
- Python Tutorial chapter 9: Classes

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Defining functions

If you find yourself running the same codes again and again with different inputs, it is time to define them as a *function*.

Here is a simple example:

In [2]:
```
def square(x):
    """Compute x*x"""
    # result returned
    return x*x
```

In [3]:
```
square(5)
```

Out[3]:
```
25
```

In [4]:
```
a = np.array([1, 2, 3])
# input `x` can be anything for which `x*x` is valid
square(a)
```

Out[4]:
```
array([1, 4, 9])
```

The line encosed by """ """ is called a *Docstring*, which is shown by `help( )` command.

In [5]:
```
help(square)
```

```
Help on function square in module __main__:

square(x)
    Compute x*x
```

A function does not need to return anything.

In [6]:
```
def print_square(x):
    """Print x*x"""
    print(x*x)
# the end of indentation is the end of definition
print_square(a)
```

```
[1 4 9]
```

A function can return multiple values.

In [7]:
```
def square_cube(x):
    """Compute x**2 and x**3"""
    # return multiple values separated by comma
    return x**2, x**3
# results can be assigned to variables separated by comma
b, c = square_cube(a)
print(b)
print(c)
```

```
[1 4 9]
[ 1  8 27]
```

In [ ]:
```

```

Arguments and local variables

A function can take single, multiple, or no arguments (inputs).
An argumet can be required, or optional with a default value.
An argument can be specified by the position, or a keyword.

In [8]:
```
def norm(x, p=2):
    """Give the L^p norm of a vector."""
    y = abs(x) ** p
    return np.sum(y) ** (1/p)
```

In [9]:
```
a = np.array([1, 2, -2])
norm(a)  # default p=2
```

Out[9]:
```
3.0
```

In [10]:
```python
norm(a, 1)  # specify by position
```
Out[10]:
```
5.0
```

In [11]:
```python
norm(p=10, x=a)  # specify by the keywords, in any oder
```
Out[11]:
```
2.1436515674591332
```

Local and global variables

Arguments and variables assigned in a function are registered in a local *namespace*.

In [12]:
```python
y = 0  # global variable
norm(a)  # this uses `y` as local variable, y=[1, 4, 9]
print(y)  # the global variable `y` is not affected
```
```
0
```

Any *global* variables can be referenced within a function.

In [13]:
```python
def add_a(x):
    """Add a to x."""
    return x + a
a = 1  # global variable
add_a(3)  # 3 + 1
```
Out[13]:
```
4
```

To modify a global variable, it have to be declaired as `global`.

In [14]:
```python
def addto_a(x):
    """Add x to a."""
    global a
    a = a + x  # add x to a
a = 1
addto_a(3)  # a = a + 3
a
```
Out[14]:
```
4
```

You can modify an argument in a function.

In [15]:
```python
def double(x):
    """Double x"""
    x = 2 * x
    return x
double(3)
```
Out[15]:
```
6
```

In [ ]:

Scripts, modules, and packages

Before Jupyter (iPython) notebook was created, to reuse any code, you had to store it in a text file, with `.py` extension by convention. This is called a *script*.

In [16]:
```
%cat haisai.py
```

```
print('Haisai!')
```

The standard way of running a script is to type in a terminal:

```
$ python haisai.py
```

In a Jupyter notebook, you can use `%run` magic command.

In [17]:
```
%run haisai.py
```

```
Haisai!
```

You can edit a python script by any text editor.

In Jupyter notebook's `Files` window, you can make a new script as a Text file by `New` menu, or edit an existing script by clicking the file name.

In [18]:
```
%run ../untitled.py
```

```
name me!
```

A script with function definitions is called a *module*.

In [19]:
```
%cat lp.py
```

```
"""L^p norm module"""

import numpy as np

def norm(x, p=2):
    """The L^p norm of a vector."""
    y = abs(x) ** p
    return np.sum(y) ** (1/p)

def normalize(x, p=2):
    """L^p normalization"""
    return x/norm(x, p)
```

You can import a module and use its function by `module.function()`.

In [20]:
```python
import lp
```

In [21]:
```python
help(lp)
```

```
Help on module lp:

NAME
    lp - L^p norm module

FUNCTIONS
    norm(x, p=2)
        The L^p norm of a vector.

    normalize(x, p=2)
        L^p normalization

FILE
    /Users/doya/Dropbox (OIST)/Python/ComputationalMethods/lp.py
```

In [22]:
```python
a = np.array([-3, 4])
lp.norm(a)
```

Out[22]:
```
5.0
```

In [23]:
```python
lp.normalize(a, 1)
```

Out[23]:
```
array([-0.42857143,  0.57142857])
```

Caution: Python reads in a module only upon the first `import`, as popular modules like `numpy` are imorted in many modules. If you modify your module, you need to restart your kernel or call `importlib.reload()`.

In [24]:
```python
import importlib
importlib.reload(lp)
```

Out[24]:
```
<module 'lp' from '/Users/doya/Dropbox (OIST)/Python/ComputationalMe
thods/lp.py'>
```

A collection of modules are put in a directory as a *package*.

In [25]:
```
# see how matplotlib is organized
%ls ~/anaconda/lib/python3.6/site-packages/matplotlib
```

```
__init__.py                         fontconfig_pattern.py
__pycache__/                        ft2font.cpython-36m-darwin.so*
_cm.py                              gridspec.py
_cm_listed.py                       hatch.py
_cntr.cpython-36m-darwin.so*        image.py
_color_data.py                      legend.py
_contour.cpython-36m-darwin.so*     legend_handler.py
_delaunay.cpython-36m-darwin.so*    lines.py
_image.cpython-36m-darwin.so*       markers.py
_mathtext_data.py                   mathtext.py
_path.cpython-36m-darwin.so*        mlab.py
_png.cpython-36m-darwin.so*         mpl-data/
_pylab_helpers.py                   offsetbox.py
_qhull.cpython-36m-darwin.so*       patches.py
_tri.cpython-36m-darwin.so*         path.py
_version.py                         patheffects.py
afm.py                              projections/
animation.py                        pylab.py
artist.py                           pyplot.py
axes/                               quiver.py
axis.py                             rcsetup.py
backend_bases.py                    sankey.py
backend_managers.py                 scale.py
backend_tools.py                    sphinxext/
backends/                           spines.py
bezier.py                           stackplot.py
blocking_input.py                   streamplot.py
cbook.py                            style/
cm.py                               table.py
collections.py                      testing/
colorbar.py                         texmanager.py
colors.py                           text.py
compat/                             textpath.py
container.py                        ticker.py
contour.py                          tight_bbox.py
dates.py                            tight_layout.py
delaunay/                           transforms.py
docstring.py                        tri/
dviread.py                          ttconv.cpython-36m-darwin.so*
figure.py                           type1font.py
finance.py                          units.py
font_manager.py                     widgets.py
```

In [ ]:

Object Oriented Programming

Object Oriented Programming has been advocated since 1980's in order to avoid confusions and facillitate extensibility or large software development. Examples are: SmallTalk, Objective C, C++, Java,... and Python!
Major features of OOP is:

- define data structure and functions together as a *Class*
- an *instance* of a class is created as an *object*
- the data (attributes) and functions (methods) are referenced as `instance.attribute` and `instance.method()`.
- a new class can be created as a *subclass* of existing classes to inherit their attributes and methods.

In [ ]:

```
```

Defining a basic class

Definition of a class starts with
`class ClassName(BaseClass):`
and include

- definition of attributes
- `__init__()` method called when a new instance is created
- definition of other methods

The first argument of a method specifies the instance, which is named `self` by convention.

In [26]:

```
class Vector:
    """A class for vector calculation."""
    default_p = 2

    def __init__(self, arr):  # make a new instance
        self.vector = np.array(arr)     # array is registered as a vec

    def norm(self, p=None):
        """Give the L^p norm of a vector."""
        if p == None:
            p = self.default_p
        y = abs(self.vector) ** p
        return np.sum(y) ** (1/p)

    def normalize(self):
        """normalize the vector"""
        u = self.vector/self.norm()
        self.vector = u
```

A new instance is created by calling the class like a function.

In [27]:
```
x = Vector([0, 1, 2])
```

Attributes and methods are referenced by `.`

In [28]:
```
x.vector
```

Out[28]:
```
array([0, 1, 2])
```

In [29]:
```
x.norm()
```

Out[29]:
```
2.23606797749979
```

In [30]:
```
x.norm(1)
```

Out[30]:
```
3.0
```

In [31]:
```
x.default_p = 1
```

In [32]:
```
x.norm()
```

Out[32]:
```
3.0
```

In [33]:
```
x.normalize()
x.vector
```

Out[33]:
```
array([ 0.        ,  0.33333333,  0.66666667])
```

In [34]:
```
# another instance
y = Vector([0, 1, 2, 3])
```

In [35]:
```
y.norm()
```

Out[35]:
```
3.7416573867739413
```

A subclass can inherit attributes and methods of base class.

In [36]:
```
class Vector2(Vector):
    """For more vector calculation."""

    def double(self):
        u = 2*self.vector
        self.vector = u
```

In [37]:
```
z = Vector2([1, 2, 3])
z.vector
```

Out[37]:
```
array([1, 2, 3])
```

In [38]:
```
z.double()
z.vector
```

Out[38]:
```
array([2, 4, 6])
```

In [39]:
```
z.default_p
```

Out[39]:
```
2
```

In [ ]:
```

```

Exercisre

1. Functions

Define the following functions and show some sample outputs.
1) Factorial of n: $1 \times 2 \times \cdots \times n$.

In [ ]:
```
def factorial(n):
    # code
```

In [ ]:
```
factorial(3)
```

2) For a circle of radius r (default r=1), given x coordinate, return possible y coordinates (i.e., both positive and negative).

In [ ]:
```
def circley(x, r=1):
   # code
```

In [ ]:
```
circley(0.5)
```

In [ ]:
```
circley(np.sqrt(2), 2)
```

3) Any function of your interest

In [ ]:
```

```

2. Classes

1) Define the `Vector` class with the following methods and test that they work correctly.

- `norm`, `normalize`: as in the previous class (use L^p norm, with default p=2).
- `scale(s)`: multiply each component by scalar s.
- `dot(v)`: a dot product with another vector v.

In [ ]:
```
class Vector:
    """A class for vector calculation."""

    #code
```

In [ ]:
```
x = Vector([0, 1, 2])
x.vector
```

In [ ]:
```
x.scale(3)
x.vector
```

In [ ]:
```
y = Vector([1, 2, 3])
x.dot(y)
```

2) Save the class Vector as a module `vector.py`, e.g., by
`New` button in the Jupyter Files tab,
copy, paste, and make any changes to the class definition,
rename the file and save.

3) Import the module and test how it works.

In [ ]:
```
import importlib
```

In [ ]:
```

```

In [ ]:
```
x = vector.Vector([0, 1, 2])
x.vector
```

In [ ]:
```
x.norm(p=1)
```

In [ ]:
```

```

In [ ]:
```

```