Branch: master ▾ Find file Copy path

ma591spring2020 / material / functions.ipynb

 asaibab Add files via upload

4b59a22 on Feb 21

1 contributor

 Raw Blame History   

729 lines (728 sloc) 16.2 KB

Functions and modules ¶

You have already been using functions such as

- `print()`
- `l = [1,2,3]; l.sort(); l.reverse()`

Default usage

```
function(arguments separated by commas)
<outputs separated by commas> = function(arguments)
```

Writing your own functions

Consider the function

where the inputs are

- is a fixed constant (acceleration due to gravity)
- v_0 is the initial velocity
- t is the time

and the outputs are y

- the height as a function of time.

We first consider a simple implementation before discussing variations.

In [2]:
```
def yt(t,v0):
    """
    The comments here form part of the documentation.
    They are known as doc strings
    Type help(yt) or yt??
    """
    g = 9.81
    return v0*t - 0.5*g*t**2.
```

The same rules of alignment and whitespacing hold as it does for other Python commands that we have learned.

Different ways of calling the function

The first way should be obvious; the next two ways show that you can use the argument names to assign the values; the last way shows that we can interchange these arguments (within limits). We will revisit this issue because it requires closer examination.

In [3]:
```
y1 = yt(0.1, 6)
```

```
y2 = yt(0.1, v0=6)
y3 = yt(t=0.1, v0=6)
y4 = yt(v0=6, t=0.1)
```

They all give the same value which is

In [3]:
```
print([y1,y2,y3,y4])
```

```
[0.55095, 0.55095, 0.55095, 0.55095]
```

Help and doc strings

In [4]:
```
help(yt)
```

```
Help on function yt in module __main__:

yt(t, v0)
    The comments here form part of the documentation.
    They are known as doc strings
    Type help(yt) or yt??
```

In [5]:
```
yt??
```

The documentation or doc strings can be accessed as a variable and, in principle, can be edited during runtime.

In [6]:
```
print(yt.__doc__)
```

```
    The comments here form part of the documentation.
    They are known as doc strings
    Type help(yt) or yt??
```

Returning multiple arguments.

Of course, Python allows for the ability to return multiple arguments. Say, we want to return $y(t)$ and its derivative

In [7]:
```
def ytandder(t,v0):
    g = 9.81
    yt = v0*t - 0.5*g*t**2.
    ytp = v0 - g*t

    return yt, ytp
```

There are a couple of different ways to call this function.

In [8]:
```
y1, y2 = ytandder(0.5,6.)      #Explicity map each output to
a new variable
```

```
print('The function is {0} and the derivative is {1}.'.forma
t(y1,y2))
```

```
The function is 1.77375 and the derivative is 1.094999999999
9998.
```

The multiple outputs can be assigned to a single variable `y` which is a `tuple`.

In [9]:
```
y = ytandder(0.5,6.0)          #Assign it to y, which is a t
uple
print(type(y))
```

```
<class 'tuple'>
```

In [10]:
```
y1 = y[0]; y2 = y[1]
print('The function is {0} and the derivative is {1}.'.forma
t(y1,y2))
```

```
The function is 1.77375 and the derivative is 1.094999999999
9998.
```

Anonymous function handles

We can define the function y(t) as an inline function using the `lambda` keyword. This feature is useful because it allows us to succinctly program functions. Another use of this type of function is that it is in a convenient form to pass to another function as a "pointer."

In [11]:
```
g = 9.81
ytn = lambda t, v0: v0*t - 0.5*g*t**2.
```

In [12]:
```
y1 = ytn(0.1, 6)
y2 = ytn(0.1, v0=6)
y3 = ytn(t=0.1, v0=6)
y4 = ytn(v0=6, t=0.1)
```

They all give the same value which is

In [13]:
```
print([y1,y2,y3,y4])
```

```
[0.55095, 0.55095, 0.55095, 0.55095]
```

Local and global variables

Variables defined within a function are considered *local* variables and are not accessible outside of this function.

In [14]:
```
def yt3(t,v0):
    glocal = 9.81
    return v0*t - 0.5*glocal*t**2.

#In this case, glocal is a local variable and should not be
 accessible
```

```
try:
    print(glocal)
except:
    print('Cannot print because local variable')
```

```
Cannot print because local variable
```

In the above case `glocal` is a local variable and cannot be accessed outside the function. In the following case, `gglobal` is a global variable and can be accessed from any function

In [15]:
```
gglobal = 9.81
def yt3(t,v0):
    return v0*t - 0.5*gglobal*t**2.

print(yt3(0.1,6.0))
```

```
0.55095
```

Default function values

Not all function arguments have to specified, we make give default arguments. We can override these arguments, if needed.

In [16]:
```
def yt2(t, v0 = 5.):
    g = 9.81
    return v0*t - 0.5*g*t**2.
```

In [17]:
```
y1 = yt2(0.1)
print('The value is', y1)
```

```
The value is 0.45094999999999996
```

Here `v0` takes the default value which is 5. Here are three different variations.

In [18]:
```
y2 = yt2(0.1,6.)
y3 = yt2(0.1, v0 = 6.)
y4 = yt2(v0=6.,t=0.1)
```

```
[0.55095, 0.55095, 0.55095]
```

In [5]:
```
print([y2,y3,y4])
```

```
[0.55095, 0.55095, 0.55095]
```

Since the default arguments can be skipped, this makes

1. calling or invoking functions much easier.
2. makes writing documentation a lot easier.
3. enforcing backwards compatibility is a lot easier.

Here is an example of documentation from matlab's pcg: It lists explicitly how the function can be called before explaining what each input/output pair.

```
x = pcg(A,b)
```

```
pcg(A,b,tol)
pcg(A,b,tol,maxit)
pcg(A,b,tol,maxit,M)
pcg(A,b,tol,maxit,M1,M2)
pcg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = pcg(A,b,...)
[x,flag,relres] = pcg(A,b,...)
[x,flag,relres,iter] = pcg(A,b,...)
[x,flag,relres,iter,resvec] = pcg(A,b,...)
```

Here is the same function from `scipy.sparse.linalg`:

```
cg(A, b, x0=None, tol=1e-05, maxiter=None, xtype=None, M=None, c
allback=None)
```

Each input quantity and output quantity is explained in detail here (https://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.sparse.linalg.cg.html).

`*args` and `**kwargs` keyword arguments

This feature allow us to have variable number of arguments with and without keywords. Consider, first, this example.

In [19]:
```
def catch_all(a,b,*args,**kwargs):
    print('a = ', a, ', and b = ', b)  #Need a minimum of tw
o "mandatory arguments"
    print("args = ", args)        #list of variable args
    print("kwargs = ", kwargs)    #dictionary of keyword arg
s
    return                        #unnecessary but harmless
```

Now let us see the many different ways we can invoke this function

In [20]:
```
catch_all(1,2)
```

```
a =  1 , and b =  2
args =  ()
kwargs =  {}
```

In [21]:
```
catch_all(2,3,4,5,c = '1', d = '2')
```

```
a =  2 , and b =  3
args =  (4, 5)
kwargs =  {'c': '1', 'd': '2'}
```

Passing functions as arguments

The function arguments can refer to "objects" of any type and the type can change during program execution. This is known as *dynamic typing*.

In [8]:
```
#Example
```

```
def add(a,b):   return a + b
```

In [10]:
```
### Add integers
print(add(1.0,2.0))
```

```
3.0
```

In [9]:
```
### Add strings
print(add('a','b'))
```

```
ab
```