Python

Mojtaba Alaei

February 19, 2017

Content of the course

Reading Keyboard Input

The raw _ input function always returns the user input as a string object:

```
C = input('C=? ')
C = float(C)
F = (9./5)*C + 32
print F
```

The Magic eval Function: which takes a string as argument and evaluates this string as a Python expression.

```
i1 = eval(input('Give input: '))
i2 = eval(input('Give input: '))
r = i1 + i2
print('%s + %s becomes %s\nwith value %s' % \
        (type(i1), type(i2), type(r), r) )
```

eval vs exec

```python
x=2
y=3
z=eval("x+y")
print(z)
B=eval("x+y+2*z==5")
print(B)
```

```python
x=2
y=3
exec("z=x+y")
print(z)
```

Turning String Expressions into Functions

Like "eval" Function, there is "exec" function to convert expressions into functions.
But there is an easy tool which is named StringFunction in scitools:

```
# turn formula into function f(x)
>>> from scitools.StringFunction import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula)
>>> f(pi)
2.8338239229952166e-15
```

Expressions involving other independent variables than x are also possible:

```
g = StringFunction('A*exp(-a*t)*sin(omega*x)',
                   independent_variable='t',
                   A=1, a=0.1, omega=pi, x=0.5)
```

Reading from the Command Line

```
import sys
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print(F)
```

```
import sys
t = float(sys.argv[1])
v0 = float(sys.argv[2])
g = 9.81
y = v0*t - 0.5*g*t**2
print(y)
```

A Variable Number of Command-Line Arguments

```
import sys
s = 0
for arg in sys.argv[1:]:
    number = float(arg)
    s += number
print('The sum of ')
for arg in sys.argv[1:]:
    print(arg)
print('is ', s)
```

```
import sys
s = sum([float(x) for x in sys.argv[1:]])
print("The sum of %s is %s" % (' '.join(sys.argv[1:]), s))
```

The construction S.join(L) places all the elements in the list L after each other with the string S in between.

Option–Value Pairs on the Command Line

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2 \quad (1)$$

We want to write a program that takes option values like this:

example.py −−t 3 −−s0 1 −−v0 1 −−a 0.5

To do this, we will do the following commands:

- First, a parser object must be created:

```
import argparse
parser = argparse.ArgumentParser()
```

- Second, we need to add the various command-line options:

```
parser.add_argument('--v0', '--initial_velocity', type=float,
                    default=0.0, help='initial velocity')
parser.add_argument('--s0', '--initial_position', type=float,
                    default=0.0, help='initial position')
parser.add_argument('--a', '--acceleration', type=float,
                    default=1.0, help='acceleration')
parser.add_argument('--t', '--time', type=float,
                    default=1.0, help='time')
```

Option–Value Pairs on the Command Line

Third, we must read the command line arguments and interpret them:

```
args = parser.parse_args()
```

The args object we now can extract the values of the var- ious registered parameters: args.v0, args.s0, args.a, and args.t.
To evaluate s:

```
s = args.s0 + args.v0*t + 0.5*args.a*args.t**2
#or by introducing new variables so that the formula
#aligns better with the mathematical notation:
s0 = args.s0; v0 = args.v0; a = args.a; t = args.t
s = s0 + v0*t + 0.5*a*t**2
```

Handling Errors

```
import sys
if len(sys.argv) < 2:
    print('You failed to provide Celsius degrees as input '\
          'on the command line!')
    sys.exit(1) # abort because of error
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print('%gC is %.1fF' % (C, F))
```

- sys.exit(0): If no errors are found, but we still want to abort the program, sys.exit(0) is used
- sys.exit(1): Any argument different from zero signifies that the program was aborted due to an error, but the precise value of the argument does not matter so here we simply choose it to be 1

try and except

```
try:
    <statements>
except:
    <statements>
```

If something goes wrong when executing the statements in the try block, Python raises what is known as an exception. The execution jumps directly to the except block whose statements can provide a remedy for the error.

try and except example

```
import sys
try:
    C = float(sys.argv[1])
except:
    print ('You failed to provide Celsius degrees as input '\
           'on the command line!')
    sys.exit(1) # abort
F = 9.0*C/5 + 32
print('%gC is %.1fF' % (C, F))
```

Error: IndexError, ValueError, NameError, ZeroDivisionError

IndexError example:

```
>>> data = [i for i in range(1,10)]
>>> data[9]
...
IndexError: list index out of range
```

ValueError example:

```
>>> C = float('21 C')
...
ValueError: could not convert string to float: '21 C'
```

NameError example:

```
>>> print(t)
...
NameError: name 't' is not defined
```

ZeroDivisionError example:

```
>>> 2.0/0
...
ZeroDivisionError: float division
```

Error: SyntaxError, TypeError

SyntaxError example:

```
>>> forr d in data:
...
forr d in data:
     ^
SyntaxError: invalid syntax
```

TypeError example:

```
>>> 'a string'*3.14
...
TypeError: can't multiply sequence by non-int of type 'float'
```

The TypeError exception is raised because the object types involved in the multiplication are wrong (str and float).

```
>>> '--'*10 # ten double dashes = 20 dashes
'--------------------'
>>> n = 4
>>> [1, 2, 3]*n
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [0]*n
[0, 0, 0, 0]
```

Error: IndentationError

IndentationError example:

```
>>> for i in range(10):
... print(i)
  File "<stdin>", line 2
    print(i)
        ^
IndentationError: expected an indented block
```

```
>>> for i in range(10):
...     if i > 2:
...     print(i)
  File "<stdin>", line 3
    print(i)
        ^
IndentationError: expected an indented block
```

try and except and error type

```
import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print('Celsius degrees must be supplied on the command line')
    sys.exit(1) # abort execution
except ValueError:
    print ('Celsius degrees must be a pure number, '\
          'not "%s"' % sys.argv[1])
    sys.exit(1)
```

Making Modules

Make a python file called for example fibo.py:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b),
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Use Modules

Useing the module name you can access the functions:

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

or

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Executing modules as scripts

```
# Fibonacci numbers module
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b),
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

“Compiled” Python files

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called spam.pyc exists in the directory where spam.py is found, this is assumed to contain an already-“byte-compiled” version of the module spam. The modification time of the version of spam.py used to create spam.pyc is recorded in spam.pyc, and the .pyc file is ignored if these don’t match.

The Module Search Path

Interpreter searches for modules in a list of directories given by the variable sys.path.
To see sys.path:

```
import sys, pprint
pprint.pprint(sys.path)
```

You can now do one of two things:

- Place the module file in one of the folders in sys.path.
- Include the folder containing the module file in sys.path.
  - You can explicitly insert a new folder name in sys.path

    ```
    modulefolder = '../../pymodules'
    sys.path.insert(0, modulefolder)
    ```

  - Your module folders can be permanently specified in the PYTHONPATH environment variable

have some fun with python

Write the follwing code (fun.py):

```
import sys
def s(t,s0,v0,a):
    return s0+v0*t+1.0/2.0*a*t**2

init_code = ''
for statement in sys.argv[1:]:
    init_code += statement + '\n'
exec(init_code)

print(s(t,s0,v0,a))
```

Then type "python fun.py t=1 s0=2 v0=3 a=1" in the terminal