

Python Development

FOSSEE

Department of Aerospace Engineering
IIT Bombay

7 February, 2010
Day 2, Session 4

Outline

1 Tests: Getting started

2 Coding Style

3 Debugging

- Errors and Exceptions
- Strategy
- Exercise

gcd revisited!

- Open `gcd.py`

```
def gcd(a, b):  
    if a % b == 0:  
        return b  
    return gcd(b, a%b)
```

```
print gcd(15, 65)  
print gcd(16, 76)
```

- `python gcd.py`

Find lcm using our gcd module

- Open `lcm.py`

- $$lcm = \frac{a*b}{gcd(a,b)}$$

```
from gcd import gcd
def lcm(a, b):
    return (a * b) / gcd(a, b)
```

```
print lcm(14, 56)
```

- `python lcm.py`

5

4

56

Writing stand-alone module

Edit `gcd.py` file to:

```
def gcd(a, b):  
    if a % b == 0:  
        return b  
    return gcd(b, a%b)  
  
if __name__ == "__main__":  
    print gcd(15, 65)  
    print gcd(16, 76)
```

- `python gcd.py`
- `python lcm.py`

Automating tests

```
if __name__ == '__main__':  
    for line in open('numbers.txt'):  
        numbers = line.split()  
        x = int(numbers[0])  
        y = int(numbers[1])  
        result = int(numbers[2])  
        if gcd(x, y) != result:  
            print "Failed gcd test  
                    for", x, y
```

Outline

1 Tests: Getting started

2 Coding Style

3 Debugging

- Errors and Exceptions
- Strategy
- Exercise

Readability and Consistency

- Readability Counts!
Code is read more often than its written.
- Consistency!
- Know when to be inconsistent.

A question of good style

```
amount = 12.68  
denom = 0.05  
nCoins = round(amount/denom)  
rAmount = nCoins * denom
```

Style Rule #1

Naming is 80% of programming

A question of good style

```
amount = 12.68  
denom = 0.05  
nCoins = round(amount/denom)  
rAmount = nCoins * denom
```

Style Rule #1

Naming is 80% of programming

Code Layout

- Indentation
- Tabs or Spaces?
- Maximum Line Length
- Blank Lines
- Encodings

Whitespaces in Expressions

- When to use extraneous whitespaces?
- When to avoid extra whitespaces?
- Use one statement per line

Comments

- No comments better than contradicting comments
- Block comments
- Inline comments

Docstrings

- When to write docstrings?
- Ending the docstrings
- One liner docstrings

More information at PEP8:

<http://www.python.org/dev/peps/pep-0008/> 5 m

Outline

1 Tests: Getting started

2 Coding Style

3 **Debugging**

- Errors and Exceptions
- Strategy
- Exercise

Outline

- 1 Tests: Getting started
- 2 Coding Style
- 3 **Debugging**
 - **Errors and Exceptions**
 - Strategy
 - Exercise

Errors

```
In []: while True print 'Hello world'
```

```
File "<stdin>", line 1, in ?
```

```
    while True print 'Hello world'
```

^

```
SyntaxError: invalid syntax
```

Errors

```
In []: while True print 'Hello world'
```

```
File "<stdin>", line 1, in ?
```

```
    while True print 'Hello world'
```

^

```
SyntaxError: invalid syntax
```

Exceptions

```
In []: print spam
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'spam' is not defined
```

Exceptions

```
In []: print spam
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'spam' is not defined
```

Exceptions

```
In []: 1 / 0
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: integer division  
or modulo by zero
```

Exceptions

```
In []: 1 / 0
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: integer division  
or modulo by zero
```

Processing user input

```
prompt = 'Enter a number(Q to quit): '  
  
a = raw_input(prompt)  
  
num = int(a) if a != 'Q' else 0
```

What if the user enters some other alphabet?

Handling Exceptions

Python provides **try** and **except** clause.

```
prompt = 'Enter a number(Q to quit): '  
  
a = raw_input(prompt)  
try:  
    num = int(a)  
    print num  
except:  
    if a == 'Q':  
        print "Exiting ..."  
    else:  
        print "Wrong input ..."
```


Outline

1 Tests: Getting started

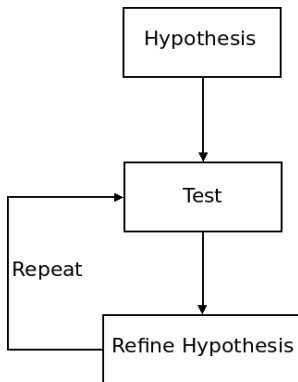
2 Coding Style

3 **Debugging**

- Errors and Exceptions
- **Strategy**
- Exercise

Debugging effectively

- **print** based strategy
- Process:



Debugging effectively

- Using `%debug` in IPython

Debugging in IPython

```
In []: import mymodule
```

```
In []: mymodule.test()
```

```
-----  
NameError      Traceback (most recent call last)
```

```
<ipython console> in <module>()
```

```
mymodule.py in test()
```

```
    1 def test():
```

```
----> 2     print spam
```

```
NameError: global name 'spam' is not defined
```

```
In []: %debug
```

```
> mymodule.py(2)test()
```

```
    0     print spam
```

```
ipdb>
```

Outline

1 Tests: Getting started

2 Coding Style

3 **Debugging**

- Errors and Exceptions
- Strategy
- **Exercise**

Debugging: Exercise

```
science = {}

for record in open('sslc1.txt'):
    fields = record.split(';')
    region_code = fields[0].strip()

    score_str = fields[6].strip()
    score = int(score_str) if score_str != 'AA'
                        else 0

    if score > 90:
        science[region_code] += 1

pie(science.values(), labels=science.keys())
savefig('science.png')
```

Summary

We have covered:

- Following and Resolving Error Messages.
- Exceptions.
- Handling exceptions
- Approach for Debugging.