CSC326 Python Imperative Core (Lec 2)

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	2011-09		JZ

Contents

1 Agenda 1

2 Invoking Python 1

3 Value, Type, Variable 1

4 Keywords 2

5 Statement 2

6 Expression 2

7 Comment 3

8 Module 3

9 Defining Functions 4

10 Conditional 5

11 Recursion 6

12 Incremental Development 7

13 Recap 8

1 Agenda

- Python imperative core
  - value, type, variable
  - expression, statement
  - module
- Incremental development w/t Python

2 Invoking Python

- Interactive mode

```
>python
Python 2.6.1 (r261, Mar 18 2009, 15:53:57)
[GCC 3.2.2] on sunos5
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Script mode

```
>python foo.py
```

3 Value, Type, Variable

- Value: Basic unit of data program manipulates
- Type:
  - A category of values (or a set of values)
  - A type itself is also a value!
- Variable: name that refers to a value

```
>>> print 'Hello, World!'
'Hello, World'
>>>print 4
4
>>> type(4)
<type 'int'>
>>> type('4')
<type 'str'>
>>> type('3.2')
<type 'float'>
>>> message = 'hi'
>>> n = 17
>>> pi = 3.14
>>> print n
17
>>> print pi
3.14
>>> type(pi)
<type 'float'>
```

Note
String could be single quoted and double quoted

```
To print "hey"
```

In C

```
  printf ("\"hey\" );
```

```
print '"hey"'
```

- No *;* after a statement!
- Hex/Oct number representation the same as C

```
>>>zipcode = 02492
```

```
SyntaxError: invalid token
```

4 Keywords

- 31 keywords

5 Statement

- print statement (obsolete for 3.0)
- assignment
- more to come

6 Expression

- functions (same as C)
  - type(.) was a function
  - function application
  - function composition a(b(v1))
- operators
  - special symbols representing builtin functions
  - apply on operands (unary or binary)
  - arithmetic: — + - **/ ^ % (just like C) — 5***2: exponentiation (Not in C)
  - relational: == != > < >= ⇐
  - logical: and, or, not
- operator precedence

  - ()
  - Exponentiation
  - * /
  - ...

- operator valid for strings (Not in C)
  - +: concetation
  - *: repetition

```
print 'hello' + 'world'
=>
'helloworld'

print 'Spam'*3
=>
'SpamSpamSpam'
```

- type conversion
  - foo(v), where foo is a type
  - between numbers: int(3.9) ⇒ 3, float(-2.3) ⇒ -2
  - between number and string: float(*3.14*) ⇒ 3.14

7 Comment

- Single line: # (// as in C++)

```
#here is a commented statement
v = 5 #assign 4 to v
```

8 Module

- Modular programming
  - Encapsulate functions and variables
  - Prevent name conflit with . notation
  - Similar to C++ namespace
  - Build Library to extend functionality: Batteries!

```
>>> import math
>>> type( math )
<type 'module'>
>>> print math
<module 'math' from 'blahblahblah./math.so'>
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10( ratio )
```

Note
math is a value of type module!

9 Defining Functions

- We see how to use functions
- Let's create our own

```
def print_lyrics() :
    print "I am a lumberjack, and I am OK."
    print "I sleep all night and I work all day."
```

Note

- def is a keyword
- paranthesis enclose function parameters
- header (first line end with colon)
- body
- end a function with empty line (if interactive mode)
- function is nothing but a value

Important

- body has to be indented!
- use space, not tab to avoid confusion
- No curly braces! (if indented anyway for readibility, why the waste?)

```
>>> print print_lyrics
<function print_lyrics at 0xb8e99e9c>
>>> print type(print_lyrics)
<type 'function'>
>>> print_lyrics()
```

- Function can call functions

```
def repeat_lyrics() :
    print_lyrics()
    print_lyrics()
```

- Function taking parameters

```
def print_twice( bruce ) :
    print bruce
    print bruce
```

- Local variables

```
def cat_twice( part1, part2 ) :
    cat = part1 + part2
    print_twice( cat )
```

- variables do NOT need to be declared at all
- variables types are inferred
- However, has to be **DEFINED** (assigned a value) before **USED**
- scoping rule applies (just like C)

10 Conditional

- Conditional
  - header followed by indented body
  - at least ONE statements in body
  - If you don't have one, or wants to come back later, use pass statement

```
if x > 0 :
   print 'x is positive'
```

```
if x < 0 :
   pass                  # does nothing!
if 0 < x and x < 10 :
   pass
```

```
if x % 2 :
   print 'x is even'
else :
   print 'x is odd'
```

- Chained conditional

```
if choice == 'a' :
   draw_a()
elif choise == 'b' :
   draw_b()
elif choice == 'c' :
   draw_c()
```

- Nested conditional
  - avoid if you could use chained conditional

```
if x == y :
   pass
else :
   if x < y :
      pass
   else :
      pass
```

11 Recursion

- Define base case
- Recurse
- Avoid infinite recursion!

```
def count_down( n ) :
    if n <= 0 :
        print 'Blastoff!'
    else :
        print n
        countdown( n-1 )
```

- compute factorial:
  - 0! = 1
  - n! = n (n - 1)!

```
def factorial( n ) :
    if n == 0 :
        return 1            # basecase
    else :
        recurse = factorial( n - 1 )
        result = n * recurse
        return result
```

- Add debugging

```
def factorial( n ) :
    space = ' ' * (4*n)   # amount of identation with repeat operator
        print space, 'calling ', n
    if n == 0 :
        print space, 'returning 1'
        return 1
    else :
        recurse = factorial( n - 1 )
        result = n * recurse
        print space, 'returning', result
        return result
```

- What would happen if we have factorial(1.5)
  - Add type checking

```
def factorial( n ) :
    if not isinstance(n,int) :   #runtime type checking, not in C
        return None
    if n == 0 :
        return 1
    else :
        recurse = factorial( n - 1 )
        result = n * recurse
        return result
```

Note
Contract based programming with preconditions

```
def factorial( n ) :
    assert isinstance(n,int)
    if n == 0 :
        return 1
    else :
        recurse = factorial( n - 1 )
        result = n * recurse
        return result
```

- Computing fibnacci series

```
def fibonacci( n ) :
    if n < 2 :
        return n
    else :
        recurse1 = fibonacci( n - 1 )
        recurse2 = fibonacci( n - 2 )
        return recurse1 + recurse2
```

12 Incremental Development

- We have seen how to use pass
- In general, code could be developed incrementally
- compute Euclidean disstance = sqrt($(x2\text{-}x1)^{2} + (y2\text{-}y1)2$)
  - Make it run first! (it does not have to be right)

```
def distance( x1, y1, x2, y2 ) :
    return 0.0
```

- Now you could at least use it with right parameters — Advantage of interpretive language: no recompile

```
>>> distance( 1, 2, 4, 6 )
0.0
```

- Add more code (including debugging code)

```
def distance( x1, y1, x2, y2 ) :
    dx = x2 - x1
    dy = y2 - y1
    print 'dx is', dx
    print 'dy is', dy
    return 0.0
```

- Add more code (including debugging code: scaffolding)

```
def distance( x1, y1, x2, y2 ) :
    dx = x2 - x1
    dy = y2 - y1
    print 'dx is', dx
    print 'dy is', dy
    dsquared = dx**2 + dy**2
    print 'dsquared is', dsquared
    result = math.sqrt( dsquared )
    return result
```

- Finalize (remove debug code)

```
def distance( x1, y1, x2, y2 ) :
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt( dsquared )
    return result
```

13 Recap

- Imperative core
  - We have covered a small set of Python
  - variables: to hold values
  - expression: to compute
  - statement: state transition
- Turing completeness
  - But according to Turing, it is complete
  - Infinite tape: function recursion

Important
Anything could be computed with this sub-language!

- Development
  - Interperative language: no compilation-link-run cycle
  - Coding style "enforced": intentatation is part of syntax
  - Encourage incremental development
  - interface first (contract with outside)
  - test first
  - algorithm skeleton first (use pass)
  - scaffolding: insert debug code