

# Functions, ROT13, Recursion

---

29 Sep 2008

CMPT14x

Dr. Sean Ho

Trinity Western University

# Some debugging tips

- Do **hand-simulation** on your code
- Use **print** statements liberally
- Double-check for **off-by-one** errors
  - Especially in counting **loops: for, range()**
- Try a **stub** program first
  - General structure of full program
  - Skip over computation/processing
    - ◆ Use **dummy** values for output
- Check out the **debugger** in IDLE

# Functions in Python

- It turns out that in Python, **every** procedure returns a value
  - **def print\_usage():**  
"""Print a brief help text."""  
**print "This is how to use this program...."**
- If **no** explicit **return** statement or return without a **value**, then the special **None** value is returned
- Must use **parentheses** when invoking procedures
  - Even those **without** arguments: **print\_usage()**
  - Otherwise you get the **function object**

# Predicates: pre-/post-conditions

```
def ASCII_to_char(code):
```

```
    """Convert from a numerical ASCII code
    to the corresponding character.
    """
```

```
    return chr(code)
```

- The parameter `code` needs to be  $<128$ : either
  - State **preconditions** clearly in docstring:
    - ◆ `pre: code is an integer between 1 and 128`
    - ◆ `post: returns the corresponding character.`
  - Or code **error-checking** in the function:

- ◆ `if code >= 128:`

# Example: error-handling

```
def ASCII_to_char(code):  
    """Convert from a numerical ASCII code  
    to the corresponding character.  
  
    pre: code is an integer  
    post: returns the corresponding character  
    """"  
  
    if (code <= 0) or (code >= 128):  
        print "ASCII_to_char(): needs to be <128"  
    else:  
        return chr(code)
```

# Call-by-value and call-by-reference

- In other languages procedures can have **side effects**:  
(M2)

```
PROCEDURE DoubleThis(VAR x: INT);
```

```
BEGIN
```

```
    x := x * 2;
```

```
END DoubleThis;
```

```
numApples := 5;
```

```
DoubleThis(numApples);
```

- **Call-by-value** means that the value in the actual parameter is **copied** into the formal parameter
- **Call-by-reference** means that the formal parameter is a **reference** to the actual parameter, so it can **modify** the value of the actual parameter (side effects)

# Python is both CBV and CBR

- In **M2**, parameters are **call-by-value**
  - Unless the formal parameter is prefixed with “**VAR**”: then it's **call-by-reference**
- In **C**, parameters are **call-by-value**
  - But you can make a parameter be a “**pointer**”
- **Python** is a little complicated: roughly speaking,
  - **Immutable** objects (7, -3.5, False) are **call-by-value**
  - **Mutable** objects (lists, user-defined objects) are **call-by-reference**

# Example of CBV in Python

```
def double_this(x):
```

```
    """Double whatever is passed as a parameter."""
```

```
    x *= 2
```

```
numApples = 5
```

```
double_this(5)           # x == 10
```

```
double_this(numApples)  # x == 10
```

```
double_this("Hello")    # x == "HelloHello"
```

- `double_this()` has the ability to modify the **global** `numApples`, but it doesn't because the changes are only done to the **local** formal parameter `x`.

# A fun example: ROT13

- **Task:** Translate characters into **ROT13** one line at a time
  - **ROT13:**
    - ◆ Treat each **letter** A-Z as a **number** between 1-26,
    - ◆ Add **13** to the number and wrap-around if necessary
    - ◆ Convert back to a **letter**
    - ◆ Preserve **case**
    - ◆ Leave all non-letter characters alone
  - e.g., **ROT13 ('a') == 'n', ROT13 ('P') == 'C',**  
**ROT13 ('#') == '#'**

# ROT13: Problem restatement

## ■ Input:

- A sequence of **letters**, ending with a newline

## ■ Computation:

- Convert letter to **numerical** form
- Add **13** and wrap-around if necessary
- Convert back to **letter** form

## ■ Output:

- Print **ROT13**'d character to screen

# ROT13: convert letters to numbers

- How do we convert from a letter character to a numerical code?
  - Use `ord(char)`: `testbed` program

```
char = raw_input("Type one character: ")
print "The ASCII code for %s is %d." % \
      (char, ord(char))
```
- ASCII codes: 'A' = 65, 'Z' = 90, 'a' = 97, 'z' = 122
- Convert back with `chr(code)`

# More fun with strings

- How do we read one **character** from a **string**?
  - In Python, characters are just strings of **length 1**
  - In C, M2, etc., strings are **arrays** of characters
- **Index** into a string (more on array indexing later):
  - ◆ **name = "Golden Delicious"**
  - ◆ **name[0]** is 'G'
- **Length** of a string:
  - ◆ **len(name)** is 16
  - ◆ **name[len(name)-1]** is 's' **# (the last character)**
- **Iterate** over string:
  - ◆ **for idx in range(len(string)):**

# ROT13: Pseudocode

- Print **intro** to the user
- **For** each character in the string:
  - Convert to **ASCII** numerical code
  - If character is an **uppercase** letter,
    - ◆ Add **13** to code
    - ◆ If code is now beyond 'Z', subtract 26 (**wrap-around**)
  - Else if character is a **lowercase** letter,
    - ◆ Add **13** to code
    - ◆ If code is now beyond 'z', subtract 26 (**wrap-around**)
  - Else (any other kind of character),
    - ◆ Leave it alone
  - Convert numerical code back to **character** and print

# How to test if upper/lower case?

- Our pseudocode involves a test if the character is an **uppercase** letter or **lowercase** letter
- How to do that?

```
if (code >= ord('a')) and (code <= ord('z')):
```

```
    # lowercase
```

```
elif (code >= ord('A')) and (code <= ord('Z')):
```

```
    # uppercase
```

```
else:
```

```
    # non-letter
```

# ROT13: Stub program pseudocode

- For each character in the string:
  - Convert to **ASCII** numerical code
  - Convert back to **character**
  - **Print** ASCII code and converted character
- This **stub** program allows us to test the char<->ASCII **conversion** process and the **string indexing**
- Tackle the **ROT13** processing later

# ROT13: Stub program code

```
"""Convert to ASCII code and back."""
```

```
text = raw_input("Input text? ")
```

```
for idx in range(len(text)):
```

```
    char = text[idx]
```

```
    code = ord(char)
```

```
    char = chr(code)
```

```
    print char, code,
```

- Sample input: hiya
- Sample output: h 104 i 105 y 121 a 97

# ROT13: Full program code

```
"""Apply ROT13 encoding."""  
import sys                                # sys.stdout.write()  
  
text = raw_input("Input text? ")  
for idx in range(len(text)):              # iterate over  
    string  
        char = text[idx]  
        code = ord(char)  
        if (code >= ord('a')) and (code <= ord('z')): #  
            lower  
                code += 13  
        if code > ord('z'):                 # wraparound  
            code -= 26
```

# ROT13: Full program code, p.2

```
elif (code >= ord('A')) and (code <= ord('Z')): #  
    upper  
    code += 13  
    if code > ord('Z'): # wraparound  
        code -= 26  
    char = chr(code)  
    sys.stdout.write(char)  
print
```

<http://twu.seanho.com/python/rot13.py>

# ROT13: Results and analysis

- Input: **hiya**
  - Output: **uvln**
- Input: **uvln**
  - Output: **hiya**
- Input: **Hello World! This is a longer example.**
  - Output: **Uryyb Jbeyq! Guvf vf n ybatre rknzcyr.**
- **Generalizations/**extensions?
  - Handle multiple lines one line at a time?

# Recursion

- **Recursion** is when a function invokes itself
- Classic example: **factorial** (!)
  - $n! = n(n-1)(n-2)(n-3) \dots (3)(2)(1)$
  - $0! = 1$
- Compute **recursively**:
  - Inductive step:  $n! = n*(n-1)!$
  - Base case:  $0! = 1$
- Inductive step: **assume**  $(n-1)!$  is calculated correctly; then we can find  $n!$
- Base case is needed to tell us where to **start**

# factorial() in Python

```
def factorial(n):  
    """Calculate n!. n should be a positive  
    integer."""  
    if n == 0:                # base case  
        return 1  
    else:                    # inductive step  
        return n * factorial(n-1)
```

- Progress is made each time: `factorial(n-1)`
- Base case prevents `infinite` recursion
- What about `factorial(-1)`? Or `factorial(2.5)`?