

Review Lectures 1-7

5 Oct 2010

CMPT140

Dr. Sean Ho

Trinity Western University

Outline for today

- Quiz 2
- Some debugging tips
- Pre- and post-conditions, error handling
- Midterm review
- Functions: call-by-value vs. call-by-reference
 - *(not on the midterm)*

Quiz 2 (5min, 10pts)

- 5 program **control**/flow abstractions? [3]
- Evaluate in Python: `3-2 ** 3<- 1` [2]
 - *(Show work for partial credit!)*
- Evaluate in Python: `list(range(-4, 5, 3))` [2]
- Name and describe 3 kinds of program **documentation** (either internal or external) talked about in class. [3]

Quiz 2 answers (10pts)

- 5 program control/flow abstractions? [3]
 - Sequence, Selection, Repetition, Composition, Parallelism
 - [3pts for all 5, 2pts if got 3-4, 1pt if got 1-2]
- Evaluate in Python: $3-2 ** 3 < -1$ [2]
 - $(3 - (2**3)) < (-1) \rightarrow (3-8) < (-1) \rightarrow \text{True}$
 - [1pt if made 1 error but showed work]
- Evaluate in Python: `list(range(-4, 5, 3))` [2]
 - `[-4, -1, 2]`

Quiz 2 answers (10pts)

- Name and describe 3 kinds of program **documentation** (either internal or external) talked about in class. [3]
 - Programmer's **diary**, user **manual**, (lab **write-up**),
 - **Comments**, **docstring**, online **help**, good **identifiers** / variable names, ...

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

Pre- and post-conditions

- Each function's **docstring** should specify:
 - **Pre-condition**: for each input param, the requirements: e.g., type, valid values, ...
 - **Post-condition**: return value and any side-effects produced by the function
- These form a “**contract**” with any code which invokes the function:

```
def sphere_volume(radius):  
    """Find the volume of a sphere given the radius.  
    Pre: radius: positive int or float.  
    Post: returns the volume as a float."""
```

Pre-/post-conditions: example

```
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 integer between 1 and 128.
Post: returns corresponding char."""`
 - Or put **error-checking** code 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)
```

Problem Solving in Software

- Software development as servant leadership
- Computer scientists as toolsmiths
- Relationships: Programmer ↔ User
 - Client ↔ Designer ↔ Coder ↔ Computer
- Team roles: producer vs. director, architect vs. engineer
- Top-down problem solving: WADES
- Hardware: input, storage, proc, control, output
- Control/flow: seq., select, repet., comp., parallel

Python Basics (ch1-3)

- Types: `int`, `float`, `str`, `bool`, `tuple`, `list`
 - ADT vs. real-world implementation
- Identifiers, variables, literals, constants, operators, operands, expressions, evaluation
- Ops: `+` `-` `*` `/` `%` `//` `**` `<` `>` `==` `!=`
 - Boolean ops: `and`, `or`, `not`, shortcut
 - Operator precedence
 - Assignment ops: `+=`, `*=`, etc.
- Static vs. dynamic typing, declaring/initializing
- Importing libraries: `math`, `string`

Strings (ch4)

- Strings: 'hi', "hi", """hi"""
- Output: `print()`
- Keyboard input: `input()`, type conversion
- String operations: `+`, `*`, `len()`
 - String library: "my string".`upper()`
- Formatting strings: `%d`, `%f`, `%s`

Programs Are More Than Code

- Design Before You Code
- Documentation:
 - External: design, pseudocode, user manual
 - Internal: comments, docstring, identifiers, online help, prompts to user

If (§7.1-7.3) and Loops (ch8)

- Selection: `if`, `elif`, `else`
- Repetition:
 - `while`: `continue`, `break`, `else`
 - Sentinel loops and sentinel variables
 - Common **errors** with loops
 - `for`: **iterating** over a list
 - `range()`

Functions (ch6)

- Defining functions, invoking functions
- Return value
- Local vs. global vars (why avoid globals?)
- Formal vs. actual parameters
- Call-by-value vs. call-by-reference
- Docstring, pre/post-conditions
 - Input validation: type checking, etc.

Call-by-value, call-by-reference

- In some languages functions 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 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 parameters can be “**pointers**”
- **Python** is a bit complicated: roughly speaking,
 - **Immutable** objects (7, -3.5, False) are **call-by-value**
 - **Mutable** objects (lists, user-defined objects) are **call-by-reference**

● *(Technically, it's “call-by-object” (ref))*

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"
```

- The **global** variable **numApples** isn't modified, because the changes are only done to the **local** formal parameter **x**.