

§2.3-2.4: Problem Solving, Documentation, Style

17 Sep 2008
CMPT14x
Dr. Sean Ho
Trinity Western University

- **Quiz ch1 today**

Quiz ch1

- Get out a blank sheet of **paper**
- In the top right corner, write
 - Your **name**
 - Student **ID#**
 - **CMPT14x Quiz 1**
 - Today's **date** (17 Sep 2008)
- **Number** your answers and provide short answers as best you can
- **Closed** book, closed notes, closed laptops/calcs
 - can reopen discreetly after you're done

Quiz ch1 (20 points, 10 minutes)

- Copy this sentence and **fill in** the blanks: **[2]**
 - “Computers are t____, and computer scientists are t_____.”
- What are the five steps of **top-down** problem solving?
 - (explain the concepts) **[5]**
- What's the difference between **7**, **7.0**, and “**7**”? **[3]**
Explain.
- What is **pseudocode**, and why is it important? **[4]**
- Name three examples of hardware for **input** and three examples of hardware for **output**. **[6]**

Quiz ch1: solutions (#1-2)

- “Computers are **tools**, and computer scientists are **toolsmiths**.”
- Five steps of **top-down** problem solving:
 - **Write** everything down
 - **Apprehend** the problem
 - **Design** a solution
 - **Execute** your plan
 - **Scrutinize** the results

Quiz ch1: solutions (#1-2)

- 7 vs. 7.0 vs. "7":
 - Type: int vs. float vs. str
- Pseudocode:
 - schematic description of program code but without worrying about syntax
- Input hardware:
 - Keyboard, mouse, touchscreen, camera, microphone, game controller, laser rangefinder, ...
- Output hardware:
 - Monitor, printer, speaker, motor, etc.

What's on for today (§2.3-2.4)

- Strong typing vs. weak typing
- Steps to problem solving: WADES in more detail
- Documentation
 - External documentation: design, manuals
 - Internal documentation: comments, docstrings
- Style guidelines

Keyboard input

- You know how to **output** using `print()`
- Use `input()` to get a value from the user:
 - `balance = input("Opening balance? ")`
 - The argument is the **prompt** string
 - **Dynamic typing**: Python interprets the user's response and determines its type
 - Just pressing **Enter** w/o input gives an **error**
- You can use `raw_input()` at the end of your program to **wait** for the user to press Enter before the program finishes

Documentation



- Document your thinking at **every step**, *even the ideas that didn't work!*
 - Programmer's **diary**: log of everything
- **External** documentation: outside the program
 - User manual:
 - ◆ What user **input** is required
 - ◆ What the user should expect the program to **output**
 - ◆ **No** details about program **internals**
- **Internal** documentation: within the program
 - Descriptive variable/module **names**
 - **Comments** in the code
 - Online **help** for the user

Examples of internal documentation

- Good variable **names**: numHashes
 - Bad variable names: x, num, i
- Comments: # in Python (to end of line)
 - # loop numHashes times
 - while (counter < numHashes):
 - ◆ print "#", # no newline
 - ◆ counter = counter + 1
- Online help:
 - "Enter 'h' for online help."

Comments

- Explain the “**why**”, not the “**what**”:
 - **Bad**: `x = x + 1` `# increment x`
 - **Good**: `x = x + 1` `# do next hashmark`
- Keep comments **up-to-date**!
 - **Incorrect** comments are worse than no comments
- Comments are no substitute for **external** documentation
 - Still need a separate **design** doc, pseudocode, user manual, etc.

Docstrings

- Python convention is to create a **docstring** at the top of every module, function, class, etc.:
 - **""" Print a bunch of hashes.**

```
Nellie Hacker, CMPT140  
"""
```

```
numHashes = input("How many hashes? ")  
...
```

- **Triple-quotes**: this is a **string**, not a **comment**
- First line is a short **summary**
- Second line is **blank**, then detailed **description**
- Automated Python **tools** read docstrings to help you organize your code

- More info: <http://www.python.org/dev/peps/pep-0257/>
CMPT 14x: coding style

Style conventions

- Not hard-and-fast rules, but **flexible** conventions that make code **easier to read** and understand
- **Variable** names: `numHashes`
 - Flexible, but I prefer no underscores, and capitalize each word (“CamelCase”)
 - First letter is **lowercase**
- **File/module** names: `helloworld.py`
 - Short, all lowercase, no underscores
- **Function** names: `print_hashes()`
 - lowercase, command predicate, underscores
- More details: <http://www.python.org/dev/peps/pep-0008/>

Expressions

- An **expression** is a combination of
 - Literals, constants, and variables,
 - Using appropriate **operations** (by type)

12 - 7

numApples * 4

- A few operators we'll look at:
 - Binary: + - * / % // **
 - Comparison: == < > <= => !=
 - Boolean: **and or not** (shortcut)



Binary arithmetic operators



- `+`, `-`, `*`: addition, subtraction, multiplication
- `**`: power: `2**4 == 16`
- `/`: division: `7.0 / 2 == 3.5`
 - On two ints, returns an int (floor): `7 / 2 == 3`
 - A note about float arithmetic: `7.2 / 2 ≠ 3.6`
- `//`: floor division
 - Same as `/` for ints: `7 // 2 == 3`
 - On floats, returns floor of quotient: `7.0 // 2 == 3.0`
- `%`: modulo (remainder): `8 % 3 == 2`
 - `8 % 0 => ZeroDivisionError`

Comparison operators

- Test for quantitative equality: $2 + 3 == 5$
- Test for inequality: $2 + 3 != 4$
 - Can also use $<>$
- Comparison: $<$, $>$, $<=$, $>=$
- Test for identity: `is`, `is not`
 - $(2, 3) == ((2, 3))$, but
 - $(2, 3)$ is not $((2, 3))$

Boolean operators: shortcut

- Boolean operators: **and or not**
 - In C/C++/Java: **&& || !**
- Python's boolean operators have **shortcut semantics**:
 - Second operand is only **evaluated** if necessary
 - ◆ **(7 / 0) and False** => ZeroDivisionError
 - ◆ **False and (7 / 0)** == False
 - Doesn't raise ZeroDivisionError
 - ◆ **True or (7 / 0)** == True
 - Same thing

Type conversions

- Python is **dynamically typed**, so operators can do implicit type **conversions** to their operands:
 - $2 \text{ (int)} + 3.5 \text{ (float)} == 5.5 \text{ (float)}$
 - ◆ Plus (+) operator converts 2 (int) to 2.0 (float)
- You can manually **convert** types:
 - $\text{int}(2.7) == 2$
 - $\text{int}(\text{True}) == 1$
 - Better alternative to $\text{input}()$:
 - ◆ $\text{ageString} = \text{raw_input}(\text{"Age? "})$
 - ◆ $\text{age} = \text{int}(\text{ageString})$



Precedence

- Of the operators we've learned, the precedence order from highest (evaluated first) to lowest (evaluated last) is
 - ******
 - **Unary +, -**
 - ***, /, %, //**
 - **Binary +, -**
 - **==, !=, <>, <, >, <=, >=**
 - **Is, is not**
 - **Not**
 - **And**
 - **or**

■ Complete precedence rules at <http://docs.python.org/ref/summary.html>

Formatted output: print with %

- The built-in function print can accept a **format string**:
 - ◆ print "You have %d apples." % 7
 - Output: "You have 7 apples."
 - It can take multiple arguments:
 - ◆ print "%d apples and %d oranges." % 7, 10
 - Output: "7 apples and 10 oranges."
 - Format codes:
 - ◆ %d: integer
 - ◆ %f: float
 - ◆ %s: string

Formatting: %d, %f

- You can specify the **field width**:

- ◆ `print "%3d apples" % 5`

- Output: " 5 apples" (note two **leading spaces**)

- ◆ `print "%-3d apples" % 5`

- Output: "5 apples" (**left-aligned**: two trailing spaces)

- ◆ `print "%03d apples" % 5`

- Output: "005 apples" (**padded** with zeros)

- ◆ `print "%4.1f apples" % 5.273`

- Output: " 5.3 apples"
- **4** is the **total** field width, including the decimal
- **1** is the number of digits **after** the decimal

Review of today (§2.3-2.4)

- Steps to problem solving: **WADES** in more detail
- **Documentation**
 - **External** documentation: design, manuals
 - **Internal** documentation:
 - ◆ **Comments**
 - ◆ **Docstrings**
- **Style** guidelines
- (see `bankinterest.py` example)