

# §2.2, 2.5, 2.11: The Anatomy of an Infant Program

---

15 Sep 2008

CMPT14x

Dr. Sean Ho

Trinity Western University

# What's on for today (§2.2, 2.5, 2.11)

---

- Anatomy of a lab write-up
- Components of a baby Python program
- Modules
- Library tools
- Literals, identifiers and reserved words
- Strings, quoting, newlines

# Pseudocode

- **Pseudocode** is sketching out your design
  - **General** enough to not get tied up in details
  - **Specific** enough to translate into code
- Use the five **control** abstractions
- Usually several **iterations** of pseudocode, getting less abstract and closer to real code
- Don't worry about **syntax**; worry about **semantics**
  - Repetition can be done with `WHILE ... DO ...` or `LOOP ... UNTIL ....`:
  - Similar semantics; different syntax

# Example: add 1..20

## ■ Try again:

- Initialize sum to 0
- Initialize counter to 1
- Repeat:
  - ◆ Add counter to sum
  - ◆ Add one to counter
- Until counter = 21

## ■ Alternate version:

- Initialize sum to 0
- Initialize counter to 1
- While counter < 21, repeat:
  - ◆ Add counter to sum
  - ◆ Add one to counter

- Same semantics, different syntax
- Top-of-loop test vs. bottom-of-loop test

# Pseudocode: you try (group effort!)

- Problem: print the **largest** of a sequence of numbers
- defender = first value
- contestant = second value
- repeat:
  - if (defender > contestant):
    - ◆ (def continues)
  - if (defender < contestant):
    - ◆ defender := contestant
  - contestant gets next value in the sequence
  - until: end of sequence (no more values)
- print defender

# Writeups for Labs 1-2 *(L1 due next wk)*

- Full writeups required starting with Lab3
- Labs 1-2 can have short writeup:
  - Design (10 marks)
    - ◆ Name, student#, CMPT14x, lab section, Lab#1, date
    - ◆ Statement of the problem
    - ◆ Discussion of solution strategy
  - Code (30 marks)
    - ◆ Name, etc. again in code header
    - ◆ Well-commented code, formatted and indented
  - Output (10 marks)
    - ◆ A couple runs with different input

# Components of “helloworld.py”

```
"""A baby Python program.
```

**Module  
docstring**



```
Name: John Doe
```

```
This is a sample program.
```

```
"""
```

```
import math
```

**Import library  
modules**



```
print "Hello World!"
```

```
print "Pi =", math.pi
```

**Program  
statements**



# Modules



- A module is a **container** holding
  - **items** and information
    - ◆ Variables, functions, etc.
  - constituting **all** or **part** of an executable **program**
- **helloworld.py** is a module that is a complete executable program
- **math** is a library module from which we imported the **pi** constant
- **math.pi** is **not** a module but a name within a module

# Identifiers

- Identifiers are **names** for stuff: e.g.,
  - Libraries (“math”)
  - Functions (“print”)
  - Variables (“numApples”)
- **Identifiers** are sequences of
  - non-blank **letters** or **digits**
  - Must **start** with a letter (*underscore \_ counts as a letter*)
- OK: Great\_Googly\_Moogly, x, My21stBirthday
- Not OK: “hi ya”, h@Xz0r, 21stBirthday
- **Case sensitive!** Print ≠ print
- These are the **rules**; we'll talk about **style** tomorrow

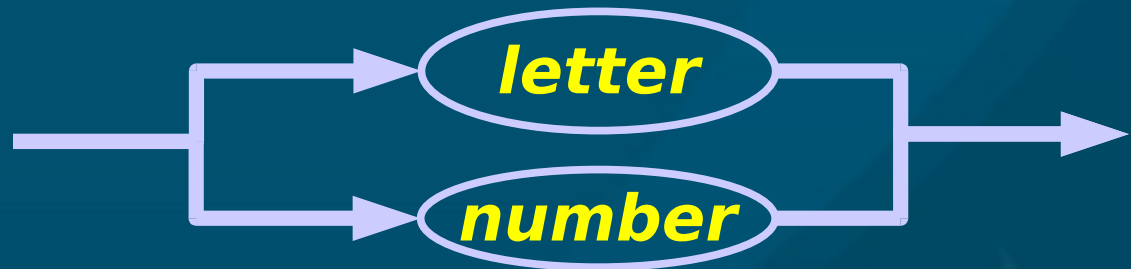
# Railroad diagram for identifiers

■ identifier

=



■ letter or  
number =



■ number = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

■ letter = {a, b, ..., z, A, B, ..., Z, \_}

# Literals vs. identifiers

- A **literal** is an entity whose name is an encoding of its value:
  - ◆ 187.3
  - ◆ "Hello World!"
  - ◆ True
- In contrast, the value of a **variable** may change even though its name stays the same:

- ◆ numApples = 5
- ◆ numApples = 6



The diagram illustrates the concept of literals. It shows two lines of code: 'numApples = 5' and 'numApples = 6'. The numbers '5' and '6' are circled in light blue. A light blue oval labeled 'literals' is positioned to the right of the second line, with a light blue arrow pointing from it to the '6' in the second line, indicating that the value is a literal.

# Reserved words

- You can **name** your modules, functions, and variables almost anything you want, **except**
- **Reserved words** (keywords): special words or markers used to outline the **structure** of a program
  - import, if, else, while, for, def, ...
  - Complete list at  
<http://docs.python.org/ref/keywords.html>



# Importing library functions

- Library functions are **building blocks**:
  - Tools that others wrote that you can use
- Functions are grouped into **libraries**:
  - If you want to use a pre-written function, you need to specify which library to **import** it from

```
import math
```

```
math.sqrt( 2 )           >>>  1.4142135623730951
```

```
math.pow( 3, 5 )        >>>  243.0
```

```
math.pi                 >>>  3.1415926535897931
```

# Python Standard Library

- Library functions provided with **every** standard Python implementation
- You still have to import them, though
- Our HelloWorld.py program used **pi** from the **math** standard library
- There are oodles of standard library functions:  
<http://docs.python.org/lib/lib.html>



# Strings and quoting



- Strings in Python can be in either 'single' or "double" quotes
- What if you want a quote mark in your string?
  - "It is I; don't be afraid"
  - 'Jesus said, "I am the way, and the truth, and the life."'
- To include a **newline** (carriage return) in string, use three double-quotes:
  - `""" This is a multi-line string.  
Even the newline is part of the  
string. """`
  - This is rather special to Python; in M2 newlines just aren't allowed in strings

# Splitting up strings: print

- print “Therefore go and”  
print “make disciples”
    - Therefore go and  
make disciples
  - print “Therefore go and”,  
print “make disciples”
    - Therefore go and make disciples
- Note trailing comma*

# Variables: names and values

- A **Python variable** is a name for a memory location, the contents of which can be changed by a program.
  - ◆ numApples
- The **assignment operator =** is the means by which the name on the left is given the value on the right.
  - ◆ numApples = numApples + 1

# Static vs. dynamic typing

- All variables have a **type**: int, float, str, bool, ...
- Some languages (C, Java, M2): **statically** typed:
  - Must **declare** the variable type ahead of time
    - ◆ **x, y: REAL;**
    - ◆ **int numApples;**
  - Can't **change** the type or assign a value of a **different** type:
    - ◆ **x := "Hello, World";**      **/\* won't work! \*/**
- But Python is **dynamically** typed:
  - ◆ **x = 5.0**
  - ◆ **x = True**      **# works in Python**

# Declaring vs. initializing

- This is only necessary for **statically-typed** languages:



- **Declare** a variable to tell the compiler the **type** of the variable:

- ◆ **VAR numApples : CARDINAL; (\* M2 \*)**

- Its value is **undefined** until it is **initialized**:

- ◆ **BEGIN**

- **numApples := 5; (\* M2 \*)**

- In a dynamically-typed language like Python, just **initialize** the variable:

- ◆ **numApples = 5 # okay in Python**

# 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

# Review of today (§2.2, 2.5, 2.11)

- Anatomy of a **lab** write-up
- Components of a baby Python **program**
- **Modules**
- **Library** tools (what are some we know already?)
- **Literals, identifiers** and **reserved** words (examples?)
- Strings, **quoting**, newlines
- Static **typing** vs dynamic typing
- Keyboard **input**