

# Ch3: Sequence, Selection (if), Repetition (while, for)

19 Sep 2007

CMPT14x

Dr. Sean Ho

Trinity Western University

- **Lab01** due tonight

# What's on for today (§2.6-3.13)

---

- Formatted output
- `abs()`, `+=`, `string.capitalize()`
- Qualified import
- Selection: `if`, `if..else..`, `if..elif..else`
- Loops: `while`
- Sentinel variables
- Loop counters
- Using mathematical **closed forms** instead of loops
- For loops

# 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 a list of 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

# String concatenation, repetition

- The plus operator (+) is overloaded to work with strings: **concatenation**

- ◆ “Hello” + “World!”      ---> “HelloWorld!”

- **Overloading** is when one operator or function can do **different** things depending on the **type** of its arguments:

- ◆ 2 + 3      --> **integer** addition

- ◆ 2 + 3.0      --> **float** addition

- ◆ “A” + “B”      --> **string** concatenation

- Python also has string **repetition**:

- ◆ “Hi!” \* 3      --> “Hi!Hi!Hi!”

# String concatenation vs. print

- **print** converts each of its arguments to a string, and puts **spaces** between them:
  - ◆ **print "Hello", "dear", "World!"**
    - ---> Hello dear World!
- String concatenation **doesn't** insert spaces:
  - ◆ **print "Hello" + "dear" + "World!"**
    - ---> HelloDearWorld!

# A few misc nifty tricks

- **Absolute** value built-in function: `abs(-5.0) --> 5.0`
- **Increment/decrement**, etc:
  - `count += 1`      # same as `count = count + 1`
  - `numApples *= 2`      # `nA = nA * 2`
  - No builtin “`++`” operator as in C++/Java
- Turn strings into **all-caps**:
  - `import string`
  - `string.upper("Hello")`      # “HELLO”

# Qualified import

- The usual way to **import** a library:  

```
import string  
string.capitalize("Hello!")
```
- Import **individual** functions from a library:  

```
from string import capitalize  
capitalize("Hello!")
```
- Or import an **entire** library (don't do this):  

```
from string import *  
capitalize("Hello!")
```
- We'll learn later about **namespaces**



# Chapter 3: Program Structure

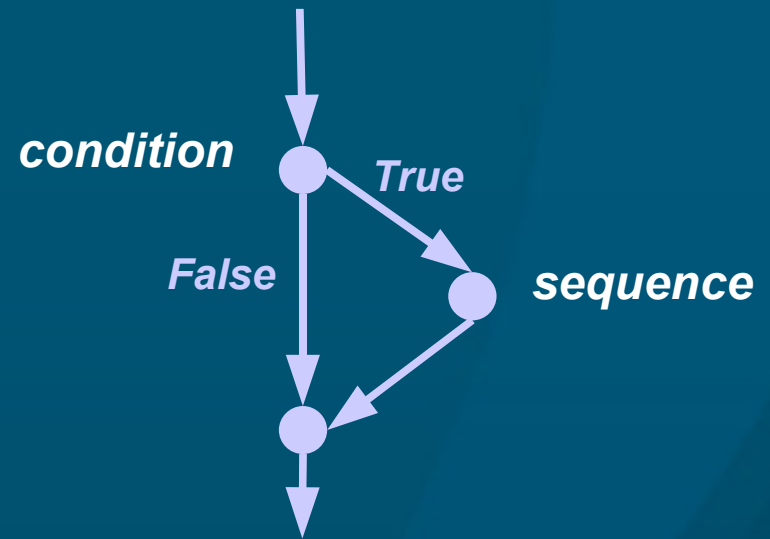
- Five basic program **structure**/flow abstractions:
  - **Sequence** (newline)
  - **Selection** (if ... elif ... else)
  - **Repetition/loops** (while, for)
  - **Composition** (subroutines)
  - **Parallelism**
- This chapter mostly covers the first **three** program structure abstractions

# Statement sequences

- A **sequence** of statements is executed in order:
  - Successive statements are not executed until the preceding statement is **completed**

```
print "Running really_slow_function() ..."  
really_slow_function()  
print "done!"
```
- Separate statements are on separate **lines**
  - **Whitespace** and **newlines** matter in Python
  - In most other languages, **semicolon (;)** separates statements, and newlines don't matter

# Simple selection: if



*if condition :*  
*statement sequence*

- Indentation (tab) indicates what's part of the statement sequence
- Condition is a **Boolean expression** evaluating to either True or False
- **Conditional execution**: if condition evaluates to False, then the statement sequence is skipped over and **not executed**

# Example using if

```
if numApples > 12:
```

```
    print "Okay, that's waay too many apples!"
```

```
print "Let's eat some apples!"
```

- Observe **indentation** (it matters in Python!)
- **Parentheses ()** not needed around the condition
  - But if the condition is complex, parentheses may be useful to clarify precedence:
    - ◆ `if (numApples > 5) and (numApples < 12)`

# Branching: if ... else ...

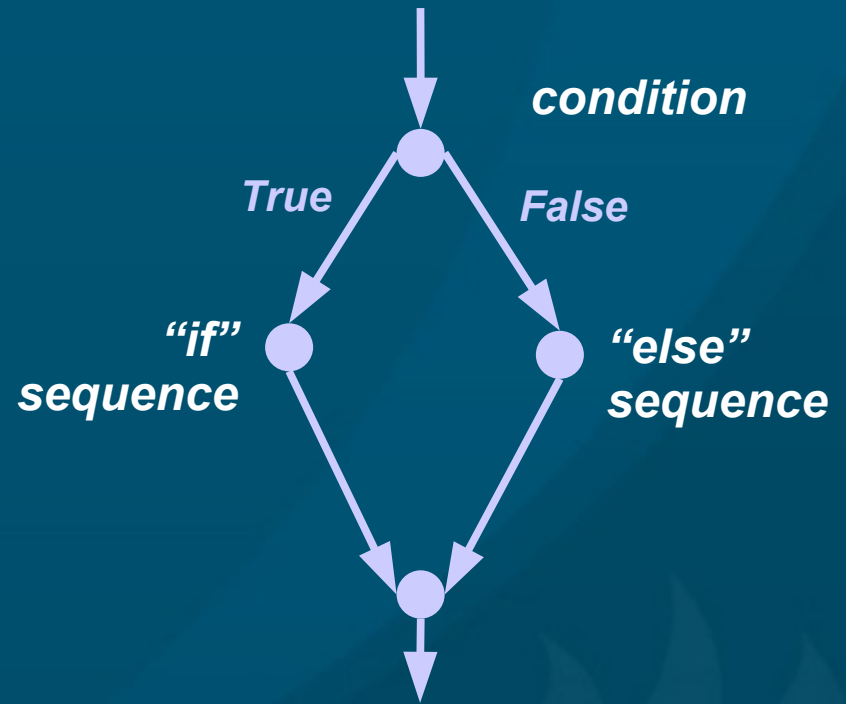
**if** *condition* :

*statement sequence*

**else** :

*statement sequence*

- Only **one** of the two statement sequences is executed



# Example using if ... else ...

```
if numFriends > 0:
```

```
    applesPerFriend = numApples / numFriends
```

```
else:
```

```
    print "Awww, you need some friends!"
```

- Would the **division** work if `numFriends == 0`?
- Will this code generate an **error** if `numFriends == 0`?

# Branching: if ... elif ... else ...

**if** condition :

statement sequence

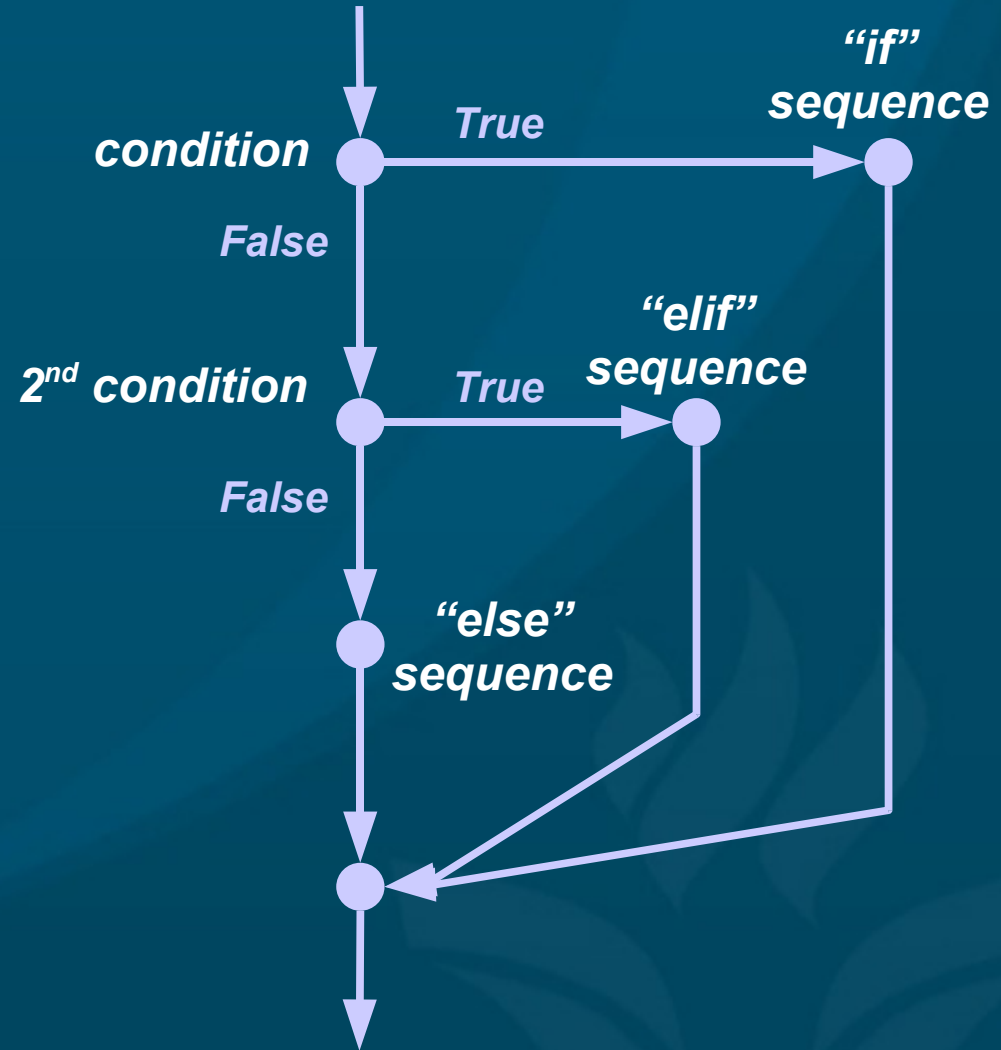
**elif** 2nd condition :

statement sequence

**else** :

statement sequence

- Only **one** of the statement sequences is executed



# Example using if ... elif ... else ...

---

```
if numFriends <= 0:
```

```
    print "Awww, you need some friends!"
```

```
elif numFriends > 30:
```

```
    print "Wow, that's a lot of friends!"
```

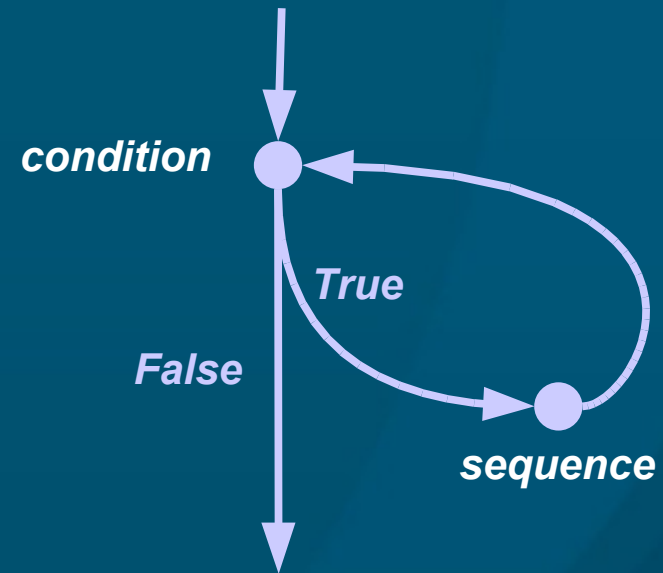
```
else:
```

```
    applesPerFriend = numApples / numFriends
```



# while loops

**while** *condition* :  
*statement sequence*



- As with “if”, *condition* is a **Boolean expression**:
  - It is evaluated **once** before entering the loop,
  - And **re-evaluated** each time through the loop:
  - Top-of-loop testing
- *Statement sequence* is run only if *condition* evaluates to True

# Sentinel variables

- A **sentinel variable** controls whether a loop continues: the loop only exits when the sentinel variable has a certain value

```
answer = 0
```

```
while answer != 4:
```

```
    answer = input("Math quiz: 2 + 2 = ")
```

- Sentinel variable is **answer**
- Sentinel value is **4**

# Counting loops

- A common form of loop uses a **counter**:

```
counter = 1
```

```
while counter <= max:
```

```
    sum = sum + counter
```

```
    counter = counter + 1
```

- What if we need to **prematurely** exit this loop?

```
counter = 1
```

```
while counter <= max:
```

```
    if need_to_exit_early():
```

```
        counter = max + 1
```

```
    ...
```

# Closed forms instead of loops

- Sometimes with a bit of thought we can replace a loop with a single **mathematical equation**
  - “Work smarter, not harder”
- Example: Add the first  $n$  integers  $>0$

```
sum = 0
```

```
counter = 1
```

```
while counter <= n:
```

```
    sum = sum + counter
```

```
    counter = counter + 1
```

```
print “Sum is %d.” % sum
```

# Closed form solution

- But observe the pattern:



- Each pair makes  $n+1$ ; there are  $n/2$  pairs:
- Closed form solution:

$$\text{sum} = n * (n+1) / 2$$

- ◆ (If  $n$  is type int, does the  $/$  cause problems?)

# while loops: continue

- You can prematurely go to the next iteration of a while loop by using **continue**:
  - ◆ **counter = 0**
  - ◆ **while counter < 5:**
    - **counter += 1**
    - **if counter == 3:**
      - **continue**
    - **print counter,**
- **Output:**
  - ◆ 1 2 4 5

# while loops: break

- You can quit a while loop early by using **break**:

- ◆ **counter = 0**

- ◆ **while counter < 5:**

- **counter += 1**
- **if counter == 3:**
  - **break**
- **print counter,**

- **Output:**

- ◆ 1 2

# while loops: else

- The optional **else** clause of a while loop is executed when the loop condition is False:
  - ◆ **counter = 0**
  - ◆ **while counter < 5:**
    - **counter += 1**
    - **print counter,**
  - ◆ **else:**
    - **print “Loop is done!”**
- Output:
  - ◆ 1 2 3 4 5 Loop is done!



# while loops: break skips else

- If the loop is exited via **break**, the **else** clause is not performed:

- ◆ **counter = 0**
- ◆ **while counter < 5:**
  - **counter += 1**
  - **if counter == 3:**
    - **break**
  - **print counter,**
- ◆ **else:**
  - **print "Loop is done!"**

- **Output:**

- ◆ **1 2**

# Common errors with loops

- Print squares from  $1^2$  up to  $10^2$ :
  - ◆ `counter = 0`
  - ◆ `while counter < 10:`
    - `print counter*counter,`
- What's wrong with this loop?
- Always make sure progress is being made in the loop!

# Common errors with loops

- Count from 1 up to 10 by twos:

- ◆ `counter = 1`
- ◆ `while counter != 10:`

- `print counter,`
- `counter += 2`

- What's wrong with this loop?  
How would you fix it?

- ◆ `counter = 1`
- ◆ `while counter < 10:`

- `print counter,`
- `counter += 2`

# Common errors with loops

- Count from 1.1 up to 2.0 in increments of 0.1:
  - ◆ `counter = 1.1`
  - ◆ `while counter != 2.0:`
    - `print counter,`
    - `counter += 0.1`
- Seems like it should work, but it might not due to inaccuracies in floating-point arithmetic
  - ◆ `counter = 1.1`
  - ◆ `while counter < 2.0:`
    - `print counter,`
    - `counter += 0.1`

# for loops

- Since many while loops are **counting** loops, the **for** loop is an easy construct that prevents many of these errors
- **Syntax:**
  - ◆ **for target in expression list :**
    - *Statement sequence*
- **Example:**
  - ◆ **for counter in (0, 1, 2, 3, 4):**
    - **print counter,**
  - **Output:**
    - ◆ 0 1 2 3 4
- for loops can also take an **else** sequence, like while loops

# range()

- The built-in function `range()` produces a list suitable for use in a for loop:
  - ◆ `range(10)` ----> `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - ◆ Note **0-based**, and doesn't include **end** of range
  - Specify **starting** value:
    - ◆ `range(1, 10)` ----> `[1, 2, 3, 4, 5, 6, 7, 8, 9]`
  - Specify **increment**:
    - ◆ `range(10, 0, -2)` ----> `[10, 8, 6, 4, 2]`
- Technically, `range()` returns a **list** (mutable), rather than a **tuple** (immutable). We'll learn about lists and mutability later.

# for loop examples

- Print squares from  $1^2$  up to  $10^2$ :
  - ◆ `for counter in range(1, 11):`
    - `print counter * counter,`
- for loops can iterate over other **lists**:
  - ◆ `for appleVariety in ("Fuji", "Braeburn", "Gala"):`
    - `print "I like", appleVariety, "apples!"`
- Technically, the for loop uses an **iterator** to get the next item to loop over. Iterators are beyond the scope of CMPT140/145.

# Review of today (§2.6-3.13)

- Formatted output
- `abs()`, `+=`, `string.capitalize()`
- Qualified import
- Selection: `if`, `if..else..`, `if..elif..else`
- Loops: `while`
- Sentinel variables
- Loop counters
- Using mathematical **closed forms** instead of loops
- For loops



# TODO items

---

- Lab01 due by midnight tonight
  - myCourses electronic turn-in should be working
  - If it doesn't work, just email your lab to me as an attachment
- HW02 due Fri:
  - 2.14 # 7 (interpret it in Python), 11, 13
- Lab02 due next Wed