

Control Structures: if, while, for

28 Sep 2010

CMPT140

Dr. Sean Ho

Trinity Western University

Outline for today

- **Formatting** strings
- `abs()`, `+=`, `string.capitalize()`, `print(,end=)`
- **Qualified** import
- **Selection**: `if`, `if..else..`, `if..elif..else`
- **Loops**: `while`
 - **Sentinel** variables and loop **counters**
 - Using **mathematical** closed forms instead of loops
 - `for` loops

String concat, 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!”

Format strings: %

- The % operator is also overloaded:
 - **“You have %d apples” % 7**
 - ◆ → **“You have 7 apples”**
- It can take a **list** of arguments:
 - **“%d apples and %d oranges” % (7, 10)**
 - ◆ → **“7 apples and 10 oranges”**
- Format codes:
 - %d: integer
 - %f: float
 - %s: string

Field width on %d and %f

- “%3d apples” % 5
 - “ 5 apples” (note two **leading** spaces)
- “%-3d apples” % 5
 - “5 apples” (left-aligned: two **trailing** spc)
- “%03d apples” % 5
 - “005 apples” (padded with **zeros**)
- “%4.1f apples” % 5.273
 - “ 5.3 apples”:
 - 4 is **total** field width, including decimal
 - 1 is number of digits **after** decimal

A few misc nifty tricks

- Absolute value built-in: `abs(-5.0)` → 5.0
- Increment/decrement, etc:
 - `count += 1` # `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”
- Output space instead of newline after `print()`:
 - `print("Status:", end=" ")`

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 (discouraged):

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

Program Structure

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

Statement sequences

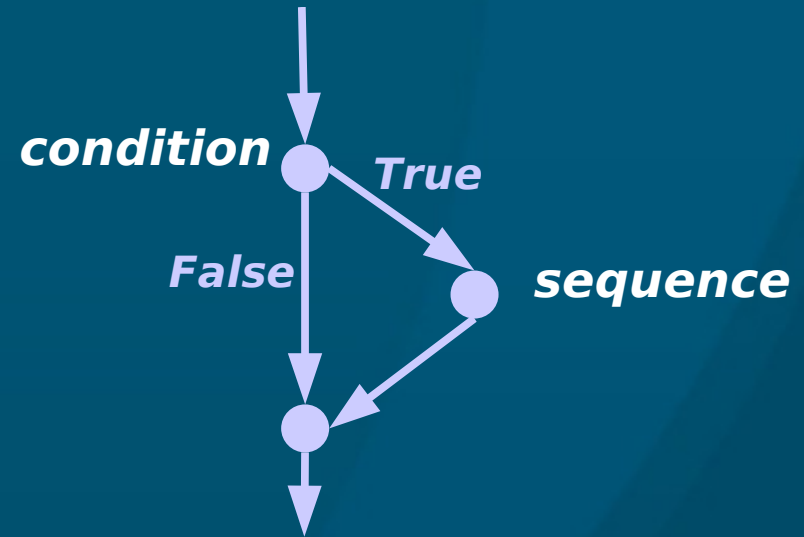
- A **sequence** of statements is executed in order:
 - Successive statements are not executed until the preceding statement is **completed**

```
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: 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 condition
 - But if condition is complex, parentheses may be useful to clarify precedence:
 - **if (numApples > 5) and (numApples < 12):**
 - **Or: if 5 < numApples < 12:**

Branching: if ... else ...

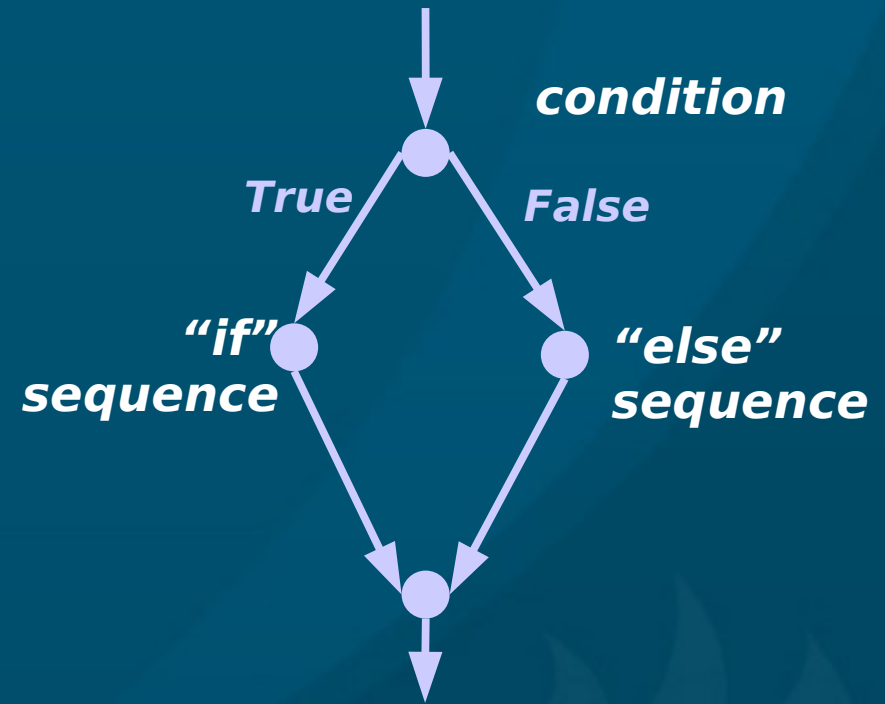
if condition :

statement sequence

else :

statement sequence

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



Example: 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

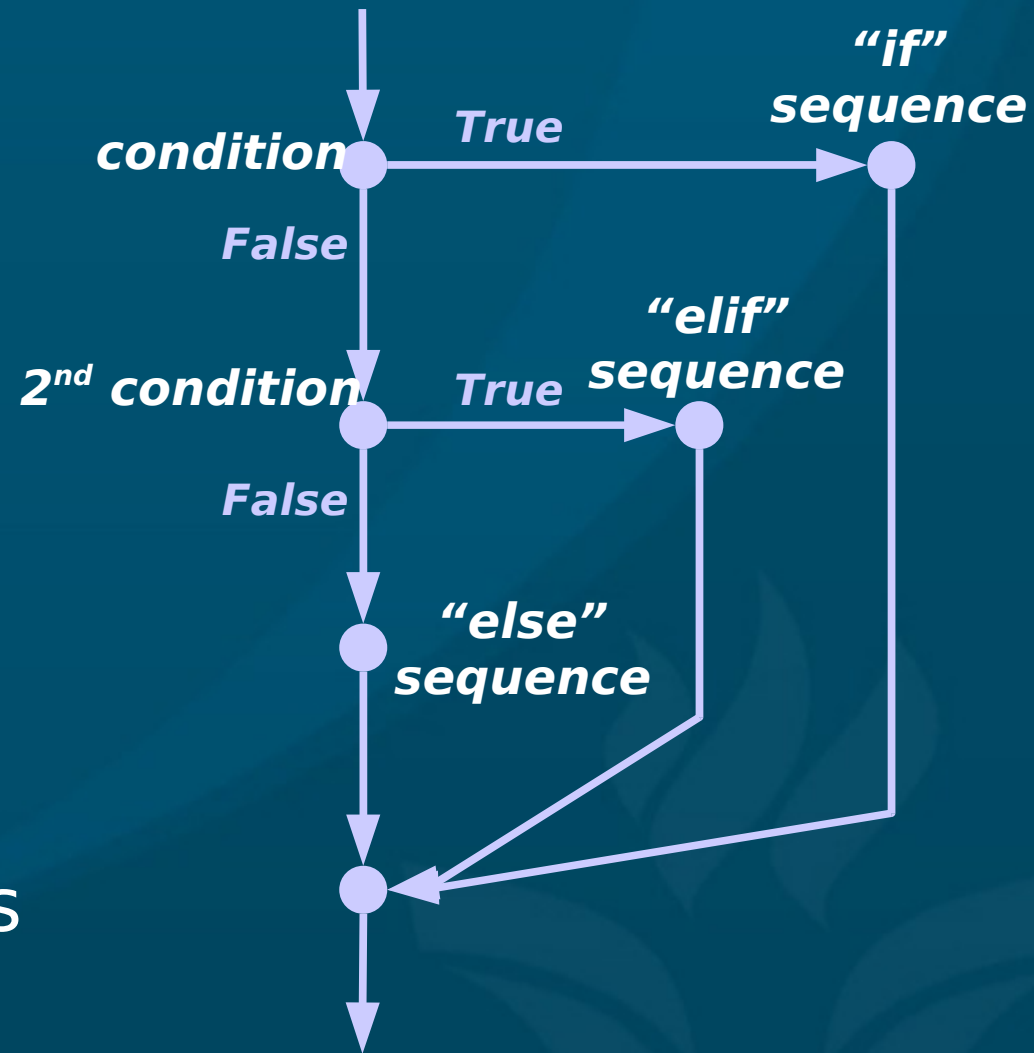
elif 2nd condition :

statement sequence

else :

statement sequence

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

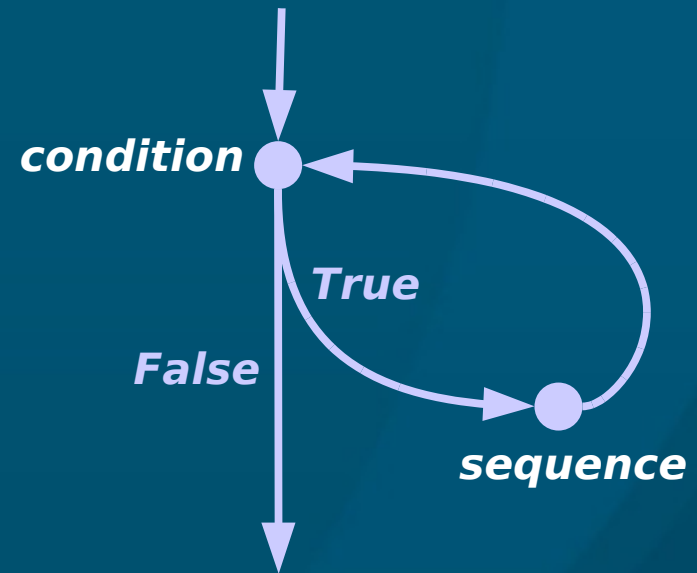


Example: if ... elif ... else ...

```
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**:
 - Evaluated **once** before entering the loop,
 - **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 = int(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 += 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 += counter
```

```
    counter += 1
```

```
print(“The sum is %d!” % sum)
```

Closed form solution

- But observe the pattern:



- Each pair makes $n+1$; there are $n/2$ pairs
 - (what if n is odd?)

- Closed-form solution:

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

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, end=" ")**
- 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, end=" ")**
- 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, end=" ")**
 - ◆ **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, end=" ")**
- ◆ **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?
 - **counter** is never **incremented**!
- → 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, end=" ")`
 - `counter += 2`
- What's **wrong** with this loop? How to **fix** it?
 - ◆ `counter = 1`
 - ◆ `while counter < 10:`
 - `print(counter, end=" ")`
 - `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, end=" ")`
 - `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, end=" ")`
 - `counter += 0.1`

for loops

- Many loops do **counting**: 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, end=" ")**
 - 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 omits **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). More on this later.*

for loop examples

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