Programming for Business Computing

Conditionals (2)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Nested `if-else` statement

- An **`if`** or an **`if-else`** statement can be **nested** in an **`if`** block.
  - In this example, if both conditions are true, ***statements A*** will be executed.
  - If ***condition 1*** is true but ***condition 2*** is false, ***statements B*** will be executed.
  - If ***condition 1*** is false, ***statements C*** will be executed.
- An **`if`** or an **`if-else`** statement can be nested in an **`else`** block.
- We may do this for any level of **`if`** or **`if-else`**.

```
if condition 1:
  if condition 2:
    statements A
  else:
    statements B
else:
  statements C
```

Example of nested `if-else` statements

- Given three integers, how to find the smallest one?
- Nested **`if-else`** helps:
- Some questions:
  - What will happen if there are multiple smallest values?
  - Are there better implementations?

```
a = int(input())
b = int(input())
c = int(input())

if a <= b:
  if a <= c:
    print(a, "is the smallest")
  else:
    print(c, "is the smallest")
else:
  if b <= c:
    print(b, "is the smallest")
  else:
    print(c, "is the smallest")
```

Two different implementations

```
min = 0
if a <= b:
  if a <= c:
    min = a
  else:
    min = c
else:
  if b <= c:
    min = b
  else:
    min = c
print(min, "is the smallest")
```

```
min = c
if a <= b:
  if a <= c:
    min = a
else:
  if b <= c:
    min = b
print(min, "is the smallest")
```

Indention matters

- In Python, an **`else`** will only be paired to the **`if`** **at the same level**.
- What does the following two problems mean?

```
if a == 10:
  if b == 10:
    print("a and b are both ten.\n")
else:
  print("a is not ten.\n")
```

```
if a == 10:
  if b == 10:
    print("a and b are both ten.\n")
  else:
    print("a is not ten.\n")
```

The ternary if operator

- In many cases, what to do after an **`if-else`** selection is simple.
- The **ternary if operator** can be helpful in this case.

```
operation A if condition else operation B
```

  – If ***condition*** is true, do ***operation A***; otherwise, ***operation B***.

- Let's modify the previous example:

```
if a <= b:
  min = a if a <= c else c
else:
  min = b if b <= c else c
```

The ternary if operator

- **Parentheses are helpful** (though not needed):

```
if a <= b:
  min = a if (a <= c) else c
else:
  min = b if (b <= c) else c
```

```
if a <= b:
  min = (a if (a <= c) else c)
else:
  min = (b if (b <= c) else c)
```

- Ternary if operators can also be nested (but **not suggested**):

```
min = (a if a <= c else c) if a <= b else (b if b <= c else c)
```

```
min = (a if a <= c else c) if (a <= b) else ((b if b <= c else c))
```

The `else-if` statement

- An **`if-else`** statement allows us to respond to one condition.
- When we want to respond to more than one condition, we may put an **`if-else`** statement in an **`else`** block:

```
if a < 10:
  print("a < 10.")
else:
  if a > 10:
    print("a > 10.")
  else:
    print("a == 10.")
```

- For this situation, people typically combine the second **`if`** behind **`else`** to create an **`else-if`** statement:

```
if a < 10:
  print("a < 10.")
elif a > 10:
  print("a > 10.")
else:
  print("a == 10.")
```

The `else-if` statement

- An **`else-if`** statement is generated by using two nested **`if-else`** statements.
- It is logically fine if we do not use **`else-if`**.
- However, if we want to respond to many conditions, using **`else-if`** greatly enhances the **readability** of our program.

```
if month == 1:
  print("31 days")
elif month == 2:
  print("28 days")
elif month == 3:
  print("31 days")
elif month == 4:
  print("30 days")
elif month == 5:
  print("31 days")
# ...
```

```
if month == 1:
  print("31 days")
else:
  if month == 2:
    print("28 days")
  else:
    if month == 3:
      print("31 days")
    else:
      # ...
```

Programming for Business Computing

Logical Operators

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Logical operators

- In some cases, the condition for an `if` statement is complicated.
  - If I am hungry **and** I have money, I will buy myself a meal.
  - If I am not hungry **or** I have no money, I will not buy myself a meal.
- We may use **logical operators** to combine multiple conditions.
- We have three logical operators: `and`, `or`, and `not`.
- There is a **precedence** rule for operators.
  - You may find the rule in the textbook.
  - You do not need to memorize them: Just use **parentheses**.

Logical operators: and

- The "and" operator operates on **two conditions**.
- It returns true if **both** conditions are true. Otherwise it returns false.
  - `(3 > 2) and (2 > 3)` returns **`False`**.
  - `(3 > 2) and (2 > 1)` returns **`True`**.
- When we use it in an **`if`** statement, the grammar is:

```
if condition 1 and condition 2:
    statements
```

Logical operators: and

- As an example:

```
a = int(input())
b = int(input())
c = int(input())

if a < b and b < c:
  print("b is in between a and c")
else:
  print("b is outside a and c")
```

Logical operators: and

- An "and" operation can replace a nested **`if`** statement.

```
a = int(input())
b = int(input())
c = int(input())

if a < b and b < c:
  print("b is in between a and c")
else:
  print("b is outside a and c")
```

```
a = int(input())
b = int(input())
c = int(input())

if a < b:
  if b < c:
    print("b is in between a and c")
  else:
    print("b is outside a and c")
else:
  print("b is outside a and c")
```

Logical operators: and

- Sometimes conditions may be combined without a logical operator:

```
if a < b < c:
  print("b is in between a and c")
```

- Nevertheless, avoid weird expressions (unless you know what you are doing):

```
if a < b < c > 10: # not good
  print("b is in between a and c")
else:
  print("b is outside a and c")
```

- Each condition must be complete by itself:

```
if b > a and < c: # error!
  print("a is between 10 and 20")
```

Logical operators: or

- The "or" operator returns true if **at least** one of the two conditions is true. Otherwise it returns false.
  - `(3 > 2) or (2 > 3)` returns **`True`**.
  - `(3 < 2) or (2 < 1)` returns **`False`**.
- When the or operator is used in an **`if`** statement, the grammar is

```
if condition 1 or condition 2:
  statements
```

Logical operators: or

- How about

```
if condition 1 or condition 2 or condition 3:
    statements
```

- How about

```
if condition 1 or condition 2 and condition 3:
    statements
```

Logical operator: not

- The "not" operator returns the **opposite** of the condition.
  - `not (2 > 3)` returns `True`.
  - `not (2 > 1)` returns `False`.
- It may be used when naturally there is nothing to do in the `if` block:

```
key = input("continue? ")

if key == "y" or key == "Y":
  print() # to avoid error
else:
  print("Game over!")
```

```
key = input("continue? ")

if not (key == "y" or key == "Y"):
  print("Game over!")
```

Programming for Business Computing

Iterations (1)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

The `while` statement

- In many cases, we want to repeatedly execute a set of codes.
- One way to implement **repetition** is to use the **`while`** statement.
- Guess what do these programs do?

```
sum = 0
i = 1

while i <= 100:
  sum = sum + i
  i = i + 1

print(sum)
```

```
# do something
exit = input("Press y or Y to exit: ")

while not (exit == "y" or exit == "Y"):
  # do something
  exit = input("Press y or Y to exit: ")
```

- **`while`** is nothing but an **`if`** that **repeats**.
  - The statements in a while block are repeated if the condition is satisfied.

Modifying loop counters

- Sometimes we need to add 1 to or subtract 1 from a **loop counter**.
- Binary **self assignment** operators (e.g., **+=**) may help.

```
sum = 0
i = 1

while i <= 100:
    sum = sum + i
    i = i + 1

print(sum)
```

```
sum = 0
i = 1

while i <= 100:
    sum = sum + i
    i += 1

print(sum)
```

```
sum = 0
i = 1

while i <= 100:
    sum += i
    i +=  1

print(sum)
```

Example

- Given an integer n, is $n = 2^k$ for some integer $k \geq 0$?

```
n = int(input())
k = 0
m = 1

while n > m:
  m *= 2
  k += 1
  # print(m, k)

if m == n:
  print(n, "is 2 to the power of", k)
```

Infinite loops

- An infinite loop is a loop that does not terminate.

```
n = int(input())
k = 0
m = 1

while n != m:
  m *= 2
  k += 1

if m == n:
  print(n, "is 2 to the power of", k)
```

- In many cases an infinite loop is a **logical error** made by the programmer.
  – When it happens, check your program.

`break` and `continue`

- When we implement a repetition process, sometimes we need to further change the flow of execution of the loop.
- A **`break`** statement brings us to **exit the loop** immediately.
- When **`continue`** is executed, statements after it in the loop are **skipped**.
  - The looping condition will be checked immediately.
  - If it is satisfied, the loop starts from the beginning again.

Example

- Which of the following programs work?

```python
n = int(input())
m = n
k = 0

while m > 1:
  if m % 2 != 0:
    break
  m //= 2
  k += 1

if m == 1:
  print(n, "is 2 to the power of", k)
```

```python
n = int(input())
m = n
k = 0

while m > 1:
  if m % 2 != 0:
    continue
  m //= 2
  k += 1

if m == 1:
  print(n, "is 2 to the power of", k)
```

`break` and `continue`

- The effect of **`break`** and **`continue`** is just on **the current level**.
  - If a **`break`** is used in an inner loop, the execution jumps to the outer loop.
  - If a **`continue`** is used in an inner loop, the execution jumps to the condition check of the inner loop.
- What will be printed out at the end of this program?

```
a = 1
b = 1
while a <= 10:
  while b <= 10:
    if b == 5:
      break
    print(a * b)
    b += 1
  a += 1
print(a)  # ?
```

Infinite loops with a break

- We may intentionally create an infinite loop and terminate it with a **break**.
  - E.g., we may wait for an "exit" input and then leave the loop with a **break**.

```
# do something
exit = input("Press y or Y to exit: ")

while not (exit == "y" or exit == "Y"):
  # do something
  exit = input("Press y or Y to exit: ")
```

```
while True:
  # do something
  exit = input("Press y or Y to exit: ")
  if exit == "y" or exit == "Y":
    break
```

Infinite loops with a `break`

- The above mentioned technique is widely used to eliminate redundant codes.

```
# do something
exit = input("Press y or Y to exit: ")

while not (exit == "y" or exit == "Y"):
  # do something
  exit = input("Press y or Y to exit: ")
```

- Redundancy introduces potential **inconsistency**.
- In some other languages, this technique is offered as a “do-while loop”.
  - In Python, just do it by yourself.

`break` and `continue`

- Using **`break`** gives a loop **multiple exits**.
  - It becomes harder to track the flow of a program.
  - It becomes harder to know the state after a loop.
- Using **`continue`** highlights the need of **getting to the next iteration**.
  - Having too many continue still gets people confused.
- Be careful **not to hurt the readability** of a program too much.

Programming for Business Computing

Iterations (2)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

The `for` statement

- Another way of implementing a loop is to use a `for` statement.

```
for variable in list:
    statements
```

- The typical way of using a for statement is:
  - ***`variable`***: A variable called the loop counter.
  - ***`list`***: A list of variables that will be "traversed."
  - ***`statements`***: The things that we really want to do.
- In each iteration, ***`variable`*** will take a value in ***`list`*** (from the first to the last).

Example

- To create a list, simply list them:

```
for i in 1, 2, 3:
  if i % 2 != 0:
    print(i)
```

```
a = 1
b = 2
c = 3

for i in a, c, b:
  print(i)
```

- A string can also be treated as a list.
  - Each character will be considered in each iteration.

```
str = "abwyz"

for i in str:
  print(i + "1")
```

`range()`

- The `range()` function is useful in creating a list of integers.
  - If n is input into `range()`, a list of integers $0, 1, 2, \ldots, n - 1$ is returned.
  - If m and n are input into `range()`, a list of integers $m, m + 1, m + 2, \ldots, n - 1$ is returned.
  - If m, n, and k are input into `range()`, a list of integers $m, m + k, m + 2k$, ... is returned, where the last integer plus k is greater than $n - 1$.
- More details about list will be introduced later in this semester.
  - For now, let's just use it in a `for` loop.

`for` vs. `while`

- Let's calculate the sum of 1 + 2 + … + 100:
  - We used **`while`**. How about **`for`**?

```
sum = 0
i = 1

while i <= 100:
  sum = sum + i
  i = i + 1

print(sum)
```

- To use **`for`**:
  - We first prepare a list of values 1, 2, …, and 100.
  - Then we sum them up.

```
sum = 0

for i in range(1, 101):
  sum = sum + i

print(sum)
```

Modifying the loop counter?

- What will be the outcome of this program?

```
sum = 0

for i in range(1, 11):
  sum = sum + i
  i = i + 10

print(sum)
```

- Manual modifications of the loop counter is of no effect!

Nested loops

- Like the selection process, **loops** can also be **nested**.
  - Outer loop, inner loop, most inner loop, etc.
- Nested loops are not always necessary, but they can be helpful.
  - Particularly when we need to handle a **multi-dimensional** case.

Nested loops: Example 1

- Please write a program to output some integer points on an (x, y)-plane like this:

(1, 1) (1, 2) (1, 3)
(2, 1) (2, 2) (2, 3)
(3, 1) (3, 2) (3, 3)

```
for x in range(3):
  x += 1
  for y in range(3):
    y += 1
    print("(" + str(x) + ", " + str(y) + ")", end = " ")
  print()
```

- Note the `end = " "` in the inner `print`.
  - It says "do not change to a new line" but "append a white space."
  - We change to a new line only in the outer loop by printing out a newline character.
- This can still be done with only one level of loop. but using a nested loop is much easier.

Nested loops: Example 2

- Please write a program to output a multiplication table:

```
for x in range(1, 5):
  for y in range(1, 5):
    print(str(x) + " * " + str(y) + " = " + str(x * y) + ";", end = " ")
  print()
```

- How would you make the lower and upper bounds flexible?
- How would you align the outputs in the same column?

Case study: single-product pricing

- We sell a product to a small town.
- The demand of this product is $q = a - bp$:
  - a is the base demand.
  - b measures the price sensitivity of the product.
  - p is the unit price to be determined.
- Let c be the unit production cost.
- Given a, b, and c, how to solve

$$\max_{p} (a - bp)(p - c)$$

  to find an optimal (profit-maximizing) price p^*?

Case study: single-product pricing

- Where there is an analytical solution $p^* = \frac{a+bc}{2b}$ (please consult the professors of your Economics/Calculus/Marketing courses), let's write a program to solve it.
- Let's assume that the price can only be an integer:

```
a = int(input("base demand = "))
b = int(input("price sensitivity = "))
c = int(input("unit cost = "))

maxProfit = 0
optimalPrice = 0
for p in range(c + 1, a // b):
  profit = (a - b * p) * (p - c)
  # print(p, profit)

  if profit > maxProfit:
    maxProfit = profit
    optimalPrice = p

print("optimal price = " + str(optimalPrice))
print("maximized profit = " + str(maxProfit))
```

Case study: single-product pricing

- Note that the profit as a function of price is first increasing and then decreasing (why?).
  - Once a price results in a profit that is lower than the maximum profit, all further prices cannot be optimal.
  - We may revise our program accordingly.

```
a = int(input("base demand = "))
b = int(input("price sensitivity = "))
c = int(input("unit cost = "))

maxProfit = 0
optimalPrice = 0
for p in range(c + 1, a // b):
  profit = (a - b * p) * (p - c)
  # print(p, profit)

  if profit > maxProfit:
    maxProfit = profit
    optimalPrice = p
  else:
    break

print("optimal price = " + str(optimalPrice))
print("maximized profit = " + str(maxProfit))
```

Good programming style

- Use the loop that makes your program the most **readable**.
- When you need to execute a loop for **a fixed number of iterations**, use a **`for`** statement with a counter declared only for the loop.
  - This also applies if you know the maximum number of iterations.
  - If the number of (maximum) number of iterations is uncertain, use **`while`**.

Programming for Business Computing

Precision Issue of Floating-point Values

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Precision can be a big issue

- Please execute the following program and try to explain the outcome:

```
import math

bad = 0
for i in range(100):
  f = pow(i, 1/2)

  if f * f != i:
    print("!!!")
    bad += 1
  else:
    print()

print("bad precision:", bad)
```

Precision can be a big issue

- Let's understand it:

```
import math

bad = 0
for i in range(100):
  f = pow(i, 1/2)
  print(i, f * f, end = " ")

  if f * f != i:
    print("!!!")
    bad += 1
  else:
    print()

print("bad precision:", bad)
```

Precision can be a big issue

- Precision can be a big issue when we use floating-point values.
- As modern computers store values in bits, most **decimal fractional numbers** can only be **approximated**.
  - 3

3	`1`	`1`	`.`	`0`	`0`	`0`	`0`
3.375	`1`	`1`	`.`	`0`	`1`	`1`	`0`
3.4375	`1`	`1`	`.`	`0`	`1`	`1`	`1`

  - 3.375
  - 3.4375
  - 3.4?
- Therefore, that **`f = pow(i, 1/2)`** does not make **`f`** storing the **exact value** of square root of **`i`**. There must be some error.

Precision can be a big issue

- Remedy: “imprecise” comparisons.

```
if abs(f * f - i) > 0.0001:
    print("!!!")
    bad += 1
else:
    print()
```

- The error tolerance can be neither too large nor too small.
  - It should be set according to the property of your own problem.
- To learn more about this issue, study *Numerical Methods*, *Numerical Analysis*, *Scientific Computing*, etc.