CS217: Computer Programming Language: Conditional

Instructor: Jakramate Bootkrajang

Outlines

- Conditional expression & Decision making
- One-way conditional
- Two-way conditional
- Multi-way conditional
- Nested conditional
- Condition simplification
- Try-Except

Motivation

- Decision making occurs a lot in our everyday life
  - What to eat ?
  - Where to travel to ?
- To decide, we do complicated information processing in our brain to come up with a decision

Conditional expression

- Naturally, information we used is in the form of conditions (lots of conditions)
- Mathematically, conditions can be expressed as boolean expression (conditional expressions)
- Boolean expression is an expression which evaluated to True or False

Conditional in Programming

- In programming, we also face with situation where we need to decide
  - To print this message or not ?
  - To continue or to quit ?
  - To perform addition or subtraction ?

The IF Statement

- In Python we can perform decision making and act accordingly using the IF statement

```
if condition(s):
□□□□statement1
□□□□statement2
          ...
```

- Condition(s) is a boolean expression which when evaluated to True, the statement(s) will be executed

The IF Statement [2]

- For example, we would have lunch at Biology's cafeteria if it is not yet noon, we could have

```
if time < 12:
    print('Let us go to Bio')
```

- Note: Don't forget the 4 spaces indentation!

The IF Statement [3]

- The IF statement is one-way conditional
- Meaning that we will perform something if the conditions evaluated to True
- Otherwise, we do nothing

Two-way conditional

- Real world is rather complicated and cruel, having only one-way conditional is quite a limitation
- We want to be able to
  - Do something if condition is True
  - Or else do some other thing

Two-way conditional [2]

- To express the alternative (the case where condition is False), Python uses the else statement

```
if condition(s):
□□□□do this if condition is
True
else:
□□□□do this if condition is
False
```

Two-way conditional [3]

- The same lunch example

```
if time < 12:
    print(‘Let us go to
Bio’)
else:
    print(‘Let us go to
OMC’)
```

Multi-way conditional

- In fact, we can extend two-way conditional into multi-way conditional with the use of the elif statement

```
if condition1(s):
□□□□do this if condition1 is
True
elif condition2(s):
□□□□do this if condition2 is
True
elif condition3(s):
□□□□do this if condition3 is
True
else:
```

Multi-way conditional [2]

- The same lunch example

```
if time < 12:
    print(‘Let us go to Bio’)
elif time < 14:
    print(‘Let us go to OMC’)
else:
    print(‘let us go to 7-11’)
```

Nested conditions

- Sometimes our situation gets very complicated such that we may need to perform another decision inside some decision
- This kind of complex decision making results in the so-called nested condition

Nested conditions [2]

- Our lunch problem

```
if time < 12:
    print('Let us go to Bio')
elif time < 14:
    print('Let us go to OMC')
else:
    if noclass:
        print('let us go to Maya')
    else:
        print('let us go to 7-11')
```

Nested if

Constructing conditions

- Conditions can be formed using basic comparison operators

operator	logical opposite
==	!=
!=	==
<	>=
<=	>
>	<=
>=	<

Constructing conditions [2]

- Single condition can be combined to form complex conditions using logical operators
  - and, or, xor, not
- For examples

```
if time < 12 and time > 10:
    Block of codes

            OR

if age < 60 or age > 18:
    Block of codes
```

Nested conditions revisited

- Nested conditional is useful for representing complex condition
- But it is also quite confusing and often leads to unintended programming bugs
- It is recommended to avoid using nested conditional if we can simplify the condition
  - e.g., using boolean algebra
  - For example using De Morgan’s laws

De Morgan’s Laws for simplifying conditions

$$\sim(a \wedge b) = \sim a \vee \sim b$$
$$\sim(a \vee b) = \sim a \wedge \sim b$$

```
not (x and y) == (not x or not y)
 not (x or y) == (not x and not y)
```

The Try-Except structure

- In reality, even if our codes perform correctly most of the time,
- There may be (rare) cases which can interupt the working of our codes
- This unforeseen error might due to users or external environments
- Careful analysis of the code might help catching these rare cases, but it takes time and is costly.

Unforseen situation

- For example, we were writing a program that asks users for their ages and acts accordingly
- We have planned our test cases that catch negative number, zero, and all positive number.
- .. which should be enough

Our program

- Displaying days old

```
age = int(input(“How old are you?”))

if age <= 0:
    print(“Are you kidding?”)
else:
    print(“You are”, age*365, “days
old”)
```

What could go wrong ?

- What if user input “ten” instead of “10” ?

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ten'
```

Try-Except to the rescue

- In Python we can use try-except structure to catch unforeseen errors
- The syntax is

```
try:
    Block of codes
    that may produce errors
except:
    Block of codes to be
    executed when error
occurs
```

Try-Except Example

- Except will catch error, and let the program continue without exiting.

```
try:
    age = int(input(“How old are you?”))
    if age <= 0:
        print(“Are you kidding?”)
    else:
        print(“You are”, age*365, “days
old”)
except:
    print(“Invalid input”)
```