ECOR 1041
Computation and Programming

Boolean Operators; Automated Testing

Carleton University

References

- *Practical Programming*, 3rd ed.
  - Chapter 5, *Making Choices*, pp. 77 - 84 (chapter introduction and “A Boolean Type” up to and excluding *Short-Circuit Evaluation)*
  - Chapter 3, *convert_to_celsius* example, p. 59 (in section “Writing and Running a Program”)
  - Chapter 15, *Testing and Debugging*, pp. 303 - 306 (chapter introduction, “Why Do You Need to Test”, and “Case Study: Testing above_freezing” up to and excluding*Testing above_freezing Using unittest)*
  - Chapter 6, pp. 110 - 112 (section “Testing Your Code Semiautomatically*)*

Lecture Objectives

- Understand how to use Boolean operators `and`, `or`, and `not`
- Learn how to automate testing by writing test functions using Boolean operators and `assert` statements

Boolean Operators

Section Objective

- Understand how to use Boolean operators `and`, `or`, and `not`

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - Boolean operators, `and`, `or`, `not`

Learning Outcomes

- Be able to use Boolean operators to combine simple comparisons into more complex comparisons

Review: Boolean Logic

- Boolean values: *true*, *false* (sometimes denoted by T, F or 1, 0)
- Boolean operations *and* (∧), *or* (∨) and *not* (¬) are summarized by a truth table:

P	***Q***	***¬ P***	***P ∧ Q***	***P ∨ Q***
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Python's Boolean Operators

- `and`, `or`, and `not`
- `not` has the highest precedence, followed by `and`, then `or`

p	***q***	**not *p***	***p* and *q***	***p* or *q***
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Boolean Operators and Conditions

- Use Boolean operators in conditional statements to combine comparisons
- Example: check if x is in the range 1 to 100, inclusive:

```
if  x >= 1 and x <= 100:
```

- We can rewrite this as:

```
if  1 <= x and x <= 100:
```

- Python lets us chain the two conditions:

```
if  1 <= x <= 100:
```

Example: sleep_in()

```
def sleep_in(weekday: bool, vacation: bool) -> bool:
    """Return True if we can sleep in; otherwise return
    False. Parameter weekday is True if today is a
    weekday. Parameter vacation is True if we are on
    vacation.
    """
```

- We can sleep in if we are on vacation, regardless of the day
- We can sleep in on the weekend, whether or not we are on vacation
  - Note: It is the weekend when weekday is False

sleep_in() Truth Table

- The calculations performed by `sleep_in()` can be summarized by this truth table:

weekday	vacation	sleep_in
True	True	True
True	False	False
False	True	True
False	False	True

sleep_in(): Version 1

```
def sleep_in(weekday: bool, vacation: bool) -> bool:
    """Return True if we can sleep in; otherwise return
    False. Parameter weekday is True if today is a
    weekday. Parameter vacation is True if we are on
    vacation.
    """
    if vacation:
        return True
    else:  # We are not on vacation, but we can sleep in
           # if it is the weekend (not a weekday)
        if weekday:
            return False
        else:
            return True
```

We do not write:

```
  if vacation == True:
```

Why?

sleep_in(): Version 2

```
def sleep_in(weekday: bool, vacation: bool) -> bool:
    """Return True if we can sleep in; otherwise return
    False. Parameter weekday is True if today is a
    weekday. Parameter vacation is True if we are on
    vacation.
    """
    if vacation:
        return True
    elif not weekday:
        # We are not on vacation, but we can sleep in
        # if it is the weekend (not a weekday)
        return True
    else:
        return False
```

sleep_in(): Version 3

- Use Boolean operators to form the condition (we can sleep in if we are on vacation or if it is the weekend)

```
def sleep_in(weekday: bool, vacation: bool) -> bool:
    """Return True if we can sleep in; otherwise return
    False. Parameter weekday is True if today is a
    weekday. Parameter vacation is True if we are on
    vacation.
    """
    if vacation or not weekday:
        return True
    else:
        return False
```

sleep_in(): Version 4

- “else” is not needed: why?

```
def sleep_in(weekday: bool, vacation: bool) -> bool:
    """Return True if we can sleep in; otherwise return
    False. Parameter weekday is True if today is a
    weekday. Parameter vacation is True if we are on
    vacation.
    """
    if vacation or not weekday:
        return True
    return False
```

sleep_in(): Version 5

- Notice that `sleep_in()` returns `True` when the condition evaluates to `True`, and returns `False` when the condition evaluates to `False`

```
def sleep_in(weekday: bool, vacation: bool) -> bool:
    """Return True if we can sleep in; otherwise return
    False. Parameter weekday is True if today is a
    weekday. Parameter vacation is True if we are on
    vacation.
    """
    return vacation or not weekday
```

Simplifying an if statement

- When an if statement has this form:

```
if condition:
    return True
else: # note: else is optional
    return False
```

- Replace it with:

```
return condition
```

Automated Testing

Section Objective

- Learn how to automate testing by writing test functions using Boolean operators and `assert` statements

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - `assert` statement
  - Unit testing
  - Regression testing

Learning Outcomes

- Understand the role of unit testing and regression testing during software development
- Be able to write simple unit test programs

Unit Testing

- Tests a single source code unit in isolation
- In Python, a unit is typically a file containing one or more closely-related functions
- Goal: show that individual units are correct, before they are integrated with other units

Testing so Far

- Manual testing of functions:
  1. Edit function definitions in a file
  2. Click Run to load the file into the Python interpreter
  3. Call a function from the shell (one test case)
  4. View and interpret the values returned by the function
  5. Repeat steps 3 and 4 until all tests performed

Manual Testing Pros and Cons

- Pros
  - O.K. for quick confidence tests
- Cons
  - Tedious if the function requires many different tests
  - Person running the tests determines if tests passed/failed (easy to overlook a failure, if there are many tests)
  - Poor for regression testing (see next slide)

Regression Testing

- Changing software (fixing bugs, redesigning code to improve maintainability, adding features, etc.) can cause:
  - New faults to be introduced into code that previously passed tests
  - Previously fixed faults to reappear

Regression Testing

- To perform regression testing, we rerun all tests to ensure that recent changes have not broken code that previously passed tests
- Manual testing is not the best way to do regression testing
  - Tedious and easy to forget to run some tests

Automating Testing

- Create a test program for each unit
- The program has one or more test cases for each function in the unit
- The program (not the person running the tests) compares actual outcomes with expected results, and determines if tests passed/failed

Case Study: Traffic Light

- Write a test program for the `traffic_light` function presented in a previous lecture
- Assume the function is in file `ECOR_1041_L8_traffic.py`
- The test program is in a separate file (`ECOR_1041_L8_test_traffic_light.py`)
- We will look at the file, line-by-line

Test Program

- Use a `from import` statement to import the function being tested into the test program:

```
from ECOR_1041_L8_traffic import traffic_light
```

- Define a function that has 3 test cases:

```
def test_traffic_light():
    assert traffic_light("red") == "green"
    assert traffic_light("green") == "yellow"
    assert traffic_light("yellow") == "red"
    print("All tests passed")
```

`assert` Statements

- `assert` is a Python keyword
- An `assert` statement has the form:

  `assert` *`expression`*

- If *`expression`* is `True`, execution continues quietly (no message is displayed)
- If *`expression`* is `False`, execution halts (by default) and an `AssertionError` is displayed

Test Program

- When used in a test program, *`expression`* compares an actual outcome of a test case with an expected result:

```
assert actual == expected
```

- Example:

```
assert traffic_light("red") == "green"
```

- Execution continues if the actual value of function call `traffic_light("red")` equals the value we expect the function to return, i.e. `"green"`

Test Program

- `print` is called to display "All tests passed" if all of the assertions are `True`
  - Otherwise, we will not reach this statement

- Finally, the test program must call the test function

```
test_traffic_light()
```

Complete Test Program

```
from ECOR_1041_L8_traffic import traffic_light

def test_traffic_light():
    """Test traffic_light."""
    assert traffic_light("red") == "green"
    assert traffic_light("green") == "yellow"
    assert traffic_light("yellow") == "red"
    print("All tests passed")

test_traffic_light()
```

Case Study: Temperature Conversion

- Write a test program for the `convert_to_celsius` function (*Practical Programming*, 3rd ed., page 59)
- Assume the function is in file `ECOR_1041_L8_temperature.py`
- The test program is in a separate file (`ECOR_1041_L8_test_temperature.py`)

Case Study: Temperature Conversion

- `convert_to_celsius` returns a `float`; e.g.,

```
>>> convert_to_celsius(75)
23.88888888888889
```

- We should never compare two floats for **equality**. Why?
  - In this case, we want to compare the actual outcome of the function call and the expected result

Case Study: Temperature Conversion

- When working with floats, to compare the actual and expected results, check if they are *almost equal*
  - Check if the absolute value of the difference between the actual and expected results is a small value:

```
assert abs(actual - expected) < epsilon
```

- Example: check if `convert_to_celsius(75)` returns a value that is "close enough" to 23.889:

```
assert abs(convert_to_celsius(75) - 23.889) < 0.001
```

Complete Test Program

```
from ECOR_1041_L8_temperature import convert_to_celsius

def test_convert_to_celsius():
    """Test convert_to_celsius."""
    assert abs(convert_to_celsius(80) - 26.667) < 0.001
    assert abs(convert_to_celsius(78.8) - 26.0) < 0.001
    assert abs(convert_to_celsius(32) - 0.0) < 0.001
    assert abs(convert_to_celsius(10.4) - -12.0) < 0.001
    print("All tests passed")

test_convert_to_celsius()
```

Do Not Do This!

- To calculate the expected result, do **not** copy/paste code from the function being tested
- Suppose `convert_to_celsius` had a bug (missing parentheses):

```
def convert_to_celsius(fahrenheit):
    return fahrenheit - 32.0 * 5 / 9
```

Do Not Do This!

- This test will pass, because the expression that calculates the expected value has the same bug as `convert_to_celsius`:

```
actual = convert_to_celsius(80)
expected = fahrenheit - 32.0 * 5 / 9 # Bug!
assert abs(actual - expected) < 0.001
```

Benefits of Unit Tests

- Simplifies integration of units into complete programs
  - Units are thoroughly tested before integration
- Documents a unit's API (Application Programming Interface)
  - Test cases show how the unit's functions should be called (similar to docstring examples)

Benefits of Unit Tests

- Can serve as specifications for a unit's functions
  - In *test-driven development*, unit tests are written before the unit is designed/coded
  - Similar to how, in the FDR, we develop docstring examples before coding the function
- Note that it is good practice for the unit tests to be developed by someone who did **not** implement the code
  - You can develop the tests as soon as the function header is available. (You do not need the function body.)

Benefits of Test Automation

- Overcomes the drawbacks of manual testing
- Facilitates change
  - regression testing is easy - just rerun the unit test programs
- Tests are collected in one place
  - useful in the long term, as the unit is maintained and/or extended

Doctests (Optional)

- *Practical Programming*, Chapter 6, presents *doctests*
- Python has a `doctest` module that can run the examples in function docstrings
- After clicking Run to load a file into Python, type in the shell:

```
>>> import doctest
>>> doctest.testmod()
```

Doctests (Optional)

- Python executes the docstring examples, and compares the values returned by the functions to the expected results shown in the examples
- To display more information during testing, run the tests like this:

```
>>> doctest.testmod(verbose = True)
```

Why Are Doctests Optional in ECOR 1041?

- For thorough testing, we often need many test cases, so the function docstrings can become very long (each test case is an example)

Why Are Doctests Optional in ECOR 1041?

- Doctests can be "brittle"
- By default, the expected output in a docstring example must match, character-by-character, the actual result returned by the function and displayed in the shell
  - it compares strings
  - doctest will report errors, even though the function returns correct values if the spacing differs from expected
    - confusing for novice programmers

Why Are Doctests Optional in ECOR 1041?

- This example in the docstring for `convert_to_celsius` passes when run by doctest:

```
>>> convert_to_celsius(75)
23.88888888888889
```

- Extra spaces in the example will cause doctest to report an error:

```
>>> convert_to_celsius(75)
      23.88888888888889
```

Recap of Learning Outcomes

Carleton University

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - Boolean operators, `and`, `or`, `not`

Learning Outcomes

- Be able to use Boolean operators to combine simple comparisons into more complex comparisons

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - `assert` statement
  - Unit testing
  - Regression testing

Learning Outcomes

- Understand the role of unit testing and regression testing during software development
- Be able to write simple unit test programs